KB

VOLUME THREE

Networks

from *How Computers Work*

KedByte Technologies Private Limited

KEDBYTE

TECHNOLOGIES PRIVATE LIMITED

HOW COMPUTERS WORK

From a Grain of Sand to Artificial Intelligence

VOLUME THREE

Networks

Signals on a wire, addresses, names, routes, encryption and diagnosis – worked from beginning to end on one real network failure, with the real traceroute, the real timeout and the real evidence.

WRITTEN BY

Shikhar Singh

Founder & Chairman

KedByte Technologies Private Limited

Part F • Chapters 23 to 34 • First Edition

HOW COMPUTERS WORK

Volume Three — Networks

Part F, chapters 23 to 34

Author	Shikhar Singh
Designation	Founder & Chairman, KedByte Technologies Private Limited

Published by	KedByte Technologies Private Limited
Imprint	KedByte Press
Edition	First Edition
Volume	3 of 5
Complete work	Nine parts, fifty-three chapters

About this volume. This is Volume 3 of five. It contains Part F of *How Computers Work*, chapters 23 to 34. Chapter numbers are those of the complete work, so a cross-reference to any chapter means the same chapter whether you are reading a single volume or the complete edition. References to chapters outside this volume point to the other volumes in the set.

Copyright © KedByte Technologies Private Limited. All rights reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other non-commercial uses permitted by copyright law.

Trademarks. All product names, company names, service marks and trademarks referred to in this book are the property of their respective owners, and are used in an editorial and educational capacity only.

Accuracy and currency. This book describes technology that changes. Specifications, prices, version numbers and industry figures were checked at the time of writing and are dated in the text wherever they are liable to change. Where experts disagree, the book says so rather than choosing a side.

Disclaimer. The information in this book is provided on an "as is" basis, for education. Neither the author nor KedByte Technologies Private Limited shall have any liability to any person or entity with respect to any loss or damage caused, or alleged to have been caused, directly or indirectly by the information contained in this book.

BEFORE YOU START

How to read this book

Every idea in this book is explained twice. First in plain language, with an everyday comparison and a worked example, so that a beginner can follow it. Then again in the correct technical vocabulary, with real units, real specifications and the exact figures an engineer would quote. The two halves are labelled, so you can read the plain pass end to end and then come back for the engineering.

Block	What it is for
PLAIN in simple words	The idea for a complete beginner. No jargon at all.
PLAIN a picture in your head	One everyday comparison, and then a line telling you exactly where that comparison stops being true.
PLAIN a worked example	Concrete numbers, worked step by step.
PLAIN what is really happening inside	The mechanism, still in easy words, but with nothing hand-waved.
TECHNICAL the engineer's version	Correct professional vocabulary, exact units, real specifications, real figures, and the commands that observe it.
WORDS remember these	Each term twice: the plain meaning, then the technical meaning.

1. Almost everything is written as short numbered lines, one idea per line, so nothing hides inside a paragraph.
2. *The honest version* marks the places where a simple explanation is not strictly true. The accurate model follows immediately.
3. The book says whether a thing is a **standard**, a **convention**, or one product's **implementation detail**.
4. Anything that goes stale is dated. Check it again before relying on it.
5. Every chapter ends with *Common wrong ideas* and a twenty-line summary. Short of time, read those two first.

The five volumes

1. **The Machine** — chapters 1 to 14
2. **Software** — chapters 15 to 22
3. **Networks** — chapters 23 to 34

4. **Building and Shipping Software** — chapters 35 to 44
5. **Intelligence, Games and Reference** — chapters 45 to 53

Contents

Part F — Networks 1

23 Networking From Zero — Signals, Ethernet, MAC, ARP and DHCP 2

23.0 What this chapter gives you 2
23.1 What a network is: two machines and a shared medium 2
23.2 Bits on a wire, physically 5
23.3 Bits on fibre and on air 8
23.4 Bandwidth, throughput, latency and jitter 11
23.5 Ethernet 14
23.6 The Ethernet frame, field by field 17
23.7 MAC addresses 20
23.8 Hubs, switches and how a switch learns 23
23.9 VLANs and spanning tree, briefly 26
23.10 ARP: from an IP address to a MAC address 29
23.11 DHCP: how the laptop got an address 33
23.12 The network interface card 36
23.13 Putting the local network together 40
23.98 Common wrong ideas 43
23.99 Chapter summary in 20 lines 44

24 IP Addresses, Private and Public, NAT and Carrier-Grade NAT 46

24.0 What this chapter gives you 46
24.1 What an IP address is 46
24.2 Networks and hosts: the split, the mask and CIDR 49
24.3 Public and private addresses 52
24.4 The other special ranges 55
24.5 Address exhaustion: why 4.29 billion was not enough 58
24.6 NAT, explained completely 60
24.7 Ports: the trick that makes NAT work 64
24.8 What NAT breaks 67
24.9 Carrier-grade NAT 70
24.10 Reading the reader's own first six hops as a story 73
24.11 Static and dynamic addresses, and how yours is chosen 75
24.12 IPv6 in brief 78
24.13 Practical: finding your own addresses on macOS 81
24.98 Common wrong ideas 84
24.99 Chapter summary in 20 lines 84

25 DNS - From a Name to an Address 86

25.0 What this chapter gives you 86
25.1 The problem DNS solves 86
25.2 The shape of the name space 89
25.3 The four kinds of participant 91
25.4 One full lookup, step by step 93
25.5 The root servers 96
25.6 Record types 98
25.7 Caching and TTL 102
25.8 Why the reader's machine uses 1.1.1.1 104

25.9 Why the same name gives different answers 107
25.10 The reader's own answer decoded 109
25.11 DNS on the wire 112
25.12 DNS security and privacy 115
25.13 Owning a name 118
25.14 Diagnosing DNS for real 120
25.98 Common wrong ideas 123
25.99 Chapter summary in 20 lines 124

26 Packets and Routing 125

26.0 What this chapter gives you 125
26.1 Why data is split up at all 125
26.2 What a packet actually is 127
26.3 The IPv4 header field by field 130
26.4 Fragmentation and MTU 132
26.5 What a router actually does, mechanically 134
26.6 The routing table 137
26.7 How routers learn routes 140
26.8 Autonomous systems 142
26.9 BGP, the protocol that glues the internet together 145
26.10 Why nobody knows the whole path in advance 148
26.11 Load balancing inside the network 150
26.12 Quality of service and the real world 152
26.13 Reading a packet capture 154
26.98 Common wrong ideas 156
26.99 Chapter summary in 20 lines 156

27 Traceroute Decoded — Reading Your Own Path 158

27.0 What this chapter gives you 158
27.1 Traceroute is a clever trick, not a feature 158
27.2 The time to live field, properly 161
27.3 The mechanism, step by step 163
27.4 The variants, and which one to use 166
27.5 Why one hop shows several addresses 168
27.6 Stars, and how to read them honestly 171
27.7 Why the times look strange 173
27.8 What a traceroute cannot tell you 176
27.9 The reader's trace, decoded hop by hop 178
27.10 Reading carrier router names 181
27.11 The trace as a story 184
27.12 What the silence after hop 12 means 186
27.13 How to run a much better traceroute 188
27.98 Common wrong ideas 190
27.99 Chapter summary in 20 lines 191

28 TCP, Ports, and Why a Timeout Is Evidence 193

28.0 What this chapter gives you 193
28.1 What IP alone gives you, and what it does not 193
28.2 TCP and UDP: two ways to use IP 195
28.3 Ports, sockets, and the four-tuple 197
28.4 The three-way handshake 200
28.5 The TCP state machine and the four-way close 202
28.6 Making TCP reliable and fast 205
28.7 Congestion control in plain words 207
28.8 The three outcomes of a connection attempt 210
28.9 What the reader's timeout actually proves 212
28.10 Why "the server is down" was ruled out 215
28.11 Reading curl -v line by line 217
28.12 The other tools that answer this question 219
28.13 Sockets from the programmer's side 221

28.98 Common wrong ideas 223
28.99 Chapter summary in 20 lines 224

29 Why Mobile Data Worked, and the Diagnostic Method 226

29.0 What this chapter gives you 226
29.1 "The site is down" and "I cannot reach the site" 226
29.2 Everything that changed when the phone used mobile data 229
29.3 The logic of a controlled experiment on a network 232
29.4 The diagnostic ladder, rung by rung 235
29.5 The reader's own fault, mapped onto the ladder 240
29.6 Cross-checks that multiply your certainty 244
29.7 Ping, honestly 246
29.8 The commands in full, for macOS 249
29.9 The decision tree for "I cannot reach a site" 252
29.10 Writing up a fault so someone else can act 254
29.11 When it is not your problem 256
29.12 Intermittent faults, the hardest kind 258
29.13 What to do next in this specific situation 261
29.98 Common wrong ideas 263
29.99 Chapter summary in 20 lines 264

30 How the Internet Was Built 265

30.0 What this chapter gives you 265
30.1 Before electricity: fire, flags and towers 265
30.2 The electric telegraph 267
30.3 The telephone network 269
30.4 Packet switching, invented three times 271
30.5 ARPANET 273
30.6 Email and the applications that came first 275
30.7 TCP/IP and the flag day 277
30.8 Other networks and the standards war 279
30.9 From research network to public internet 281
30.10 The World Wide Web 283
30.11 The browser wars 284
30.12 The dot-com boom and bust 286
30.13 Broadband and the physical internet 288
30.14 Who actually runs the internet 291
30.15 Web 1.0 to now 293
30.16 A master internet timeline 294
30.98 Common wrong ideas 296
30.99 Chapter summary in 20 lines 297

31 Wireless - WiFi, Bluetooth, Cellular and GPS 298

31.0 What this chapter gives you 298
31.1 Radio from first principles 298
31.2 Carrying information on a wave 300
31.3 The realities of radio 303
31.4 Spectrum and regulation 305
31.5 The WiFi hardware 307
31.6 WiFi history and standards 310
31.7 How WiFi actually works on the air 312
31.8 WiFi security 315
31.9 The WiFi software stack 317
31.10 Bluetooth 320
31.11 Bluetooth Low Energy 322
31.12 Bluetooth audio 324
31.13 Cellular 326
31.14 GPS and positioning 328
31.15 The other short-range radios 330
31.98 Common wrong ideas 332

31.99 Chapter summary in 20 lines 332

32 TLS, HTTPS and the Web Stack 334

32.0 What this chapter gives you 334
32.1 HTTP first, because TLS wraps it 334
32.2 The three problems TLS solves 337
32.3 The crypto building blocks, taught gently 339
32.4 The TLS handshake, in full 342
32.5 Certificates: what is actually in one 345
32.6 The chain of trust 348
32.7 What the browser verifies, and what it does NOT 351
32.8 Self-signed certificates and the reader's own router 354
32.9 When TLS is deliberately intercepted 356
32.10 HTTP versions: 1.1, 2 and 3 358
32.11 Cookies, sessions, and CORS 360
32.12 The whole journey, assembled 363
32.13 CDNs, load balancers and caching in front 365
32.98 Common wrong ideas 367
32.99 Chapter summary in 20 lines 368

33 The OSI and TCP/IP Models, Retrofitted to Your Own Session 369

33.0 What this chapter gives you 369
33.1 Why layers exist at all 369
33.2 The OSI seven-layer model 371
33.3 The TCP/IP four-layer model 374
33.4 The reader's own session, layer by layer 376
33.5 Encapsulation walked through with real numbers 382
33.6 Where the reader's failure sits in the model 385
33.7 A fault-to-layer table 387
33.8 Where the model lies 389
33.9 Why layering still earns its keep 391
33.10 The same idea outside networking 393
33.98 Common wrong ideas 394
33.99 Chapter summary in 20 lines 395

34 Firewalls, VPNs, CDNs, IPv6 and the Gaps 397

34.0 What this chapter gives you 397
34.1 localhost, 127.0.0.1 and 0.0.0.0 397
34.2 Ports, properly this time 400
34.3 Firewalls, properly 403
34.4 DROP versus REJECT 407
34.5 Middleboxes, and the end of end-to-end 410
34.6 VPNs, what they do and what they do not 413
34.7 Proxies, forward and reverse 416
34.8 Tunnelling in general 419
34.9 CDNs and anycast in depth 422
34.10 Latency versus bandwidth 427
34.11 IPv6 properly 431
34.12 Multicast, broadcast and mDNS 435
34.13 Network performance for developers 439
34.14 Cloud networking preview 443
34.98 Common wrong ideas 445
34.99 Chapter summary in 20 lines 445

KB

PART F

Networks

Signals, addresses, names, routes and encryption. Worked throughout on one real network failure.

23

CHAPTER

KB

Networking From Zero — Signals, Ethernet, MAC, ARP and DHCP

23.0 What this chapter gives you

1. You will be able to say what a network actually is, and name the four jobs every network must do no matter how it is built.
2. You will be able to explain how a bit becomes a voltage on copper, and why a plain high-low signal is not good enough.
3. You will be able to name the line codes that fix it, from Manchester in 1973 to PAM-4 today, and say what each one buys.
4. You will be able to read a cable category number and state its real bandwidth and distance limit instead of guessing.
5. You will be able to tell bandwidth, throughput, latency and jitter apart, and compute a bandwidth-delay product on paper.
6. You will be able to draw an Ethernet frame field by field with correct byte lengths, and explain where the number 1500 came from.
7. You will be able to read a MAC address, find its vendor, and say why your phone lies about it on purpose.
8. You will be able to describe how a switch learns, why a loop is a disaster, and what spanning tree does about it.
9. You will be able to walk through ARP and DHCP message by message, using your own router at 192.168.0.1 as the example.
10. You will be able to explain everything that happens on the wire before a single IP packet is sent, which is what the next six chapters build on.

This chapter opens Part F. Part F uses one real event as its running example: a real network outage on a real home broadband line in India, on a macOS laptop, where `curl -v https://github.com` printed `Trying 20.207.73.82:443...` and then sat in complete silence for fifteen seconds. We will take that event apart layer by layer across the whole of Part F. This chapter covers everything underneath the IP layer: the signal, the cable, the frame, the address, and the two small protocols that get a machine onto a network at all. IP addressing, NAT, DNS, routing, traceroute, TCP and TLS each get a full chapter of their own later in Part F. Where they come up here, we name them and move on.

23.1 What a network is: two machines and a shared medium

23.1.1 PLAIN – in simple words

1. A network is two or more machines that can send each other messages.
2. To send a message you need something in between. A wire, a fibre, or air.

3. That in-between thing is called the **medium**. It is the road the message travels on.
4. If only two machines share the medium, life is easy. Anything you send can only arrive at one place.
5. As soon as three or more machines share it, four new problems appear at once.
6. Problem one is **addressing**. If the message can reach everyone, it must say who it is for.
7. Problem two is **framing**. The medium carries a stream of signals. Something must mark where one message stops and the next starts.
8. Problem three is **error detection**. Wires pick up noise. The receiver must be able to tell that what arrived is not what was sent.
9. Problem four is **sharing**. If two machines talk at the same time on one shared medium, both messages are ruined. Somebody has to take turns.
10. Every network technology ever built is an answer to those same four problems. The answers differ. The problems do not.

23.1.2 PLAIN – a picture in your head

1. Picture a long corridor in a hostel with many rooms opening onto it.
2. If you shout from your doorway, everyone in the corridor hears you. The corridor is the shared medium.
3. So you begin with a name: "Anita, the parcel is at the desk." That is addressing. Everyone hears it. Only Anita acts on it.
4. You say the whole sentence in one go, then stop. The pause tells listeners the message has ended. That is framing.
5. If a bike goes past outside and drowns out three words, Anita asks you to repeat. That is error detection.
6. If two people shout at once, nobody can make out either. So people listen first, and wait if they hear a voice. That is sharing the medium.
7. Now picture a different arrangement: a private phone line from your room to Anita's room only. No names needed. No shouting. No waiting.
8. That is the difference between a **broadcast** medium and a **point-to-point** link.

Where this comparison breaks: a corridor is slow and forgiving. Humans hear a clash and naturally back off. On a wire, two signals overlapping do not sound wrong, they simply produce a third voltage that looks like valid data. The machine cannot notice by ear. It has to be told to check, with a checksum, or to detect the clash electrically within a fixed number of microseconds. Also, a corridor has no fixed speed. A wire does, and that fixed speed is what makes all the timing rules later in this chapter possible.

23.1.3 PLAIN – a worked example

1. Take a home with a laptop, a phone, a smart TV and a router.
2. The router is at `192.168.0.1`. That is the real address of the reader's router, and we will use it throughout this chapter.
3. All four devices are on one logical shared network, even though the laptop and phone reach it over radio and the TV over a cable.
4. The laptop wants to send something to the router. It cannot just push the bytes out and hope.
5. It must attach the router's local hardware address, so the network knows the message is for the router and not the TV. That is addressing.
6. It must wrap the bytes in a start marker and an end marker so the router knows where the message begins and ends. That is framing.
7. It must append a check number computed from the bytes, so the router can confirm nothing was corrupted. That is error detection.
8. Over radio, it must wait until the air is quiet before transmitting, because the phone might be talking. That is sharing.
9. Over the cable to the TV, none of the waiting is needed, because a modern switched cable link is private in both directions.
10. Same four problems. Two different sets of answers, chosen by the medium.

23.1.4 PLAIN - what is really happening inside

1. Networks are built in layers, and each layer solves one of those problems and hands the rest to the layer above.
2. The bottom layer is the **physical layer**. Its only job is to turn a 1 or a 0 into something the medium can carry, and back again.
3. Above it sits the **data link layer**. This is where framing, local addressing, error detection and medium sharing live.
4. Above that sits the network layer, which is where IP lives, and which we deliberately leave for the next chapter.
5. A **topology** is the shape of the connections. There are four classic shapes.
6. A **bus** is one long shared cable with every machine tapped onto it. Cheap, and one cut kills everything. This was early Ethernet.
7. A **star** has every machine wired to one central box. One cut kills one machine. This is what every home and office uses now.
8. A **ring** passes messages around a loop, each machine to the next. Token Ring and FDDI worked this way. Fibre backbones still use rings for resilience.
9. A **mesh** connects many machines to many others, so there are several paths between any two. The internet core is a mesh. So is a modern WiFi mesh kit.
10. In practice a home network is a star of stars: devices star into the router, and the router is one point in the ISP's tree.

23.1.5 TECHNICAL - the engineer's version

1. The reference model in textbooks is the OSI seven-layer model, published as ISO/IEC 7498-1, with the first version issued in 1984.
2. Real systems follow the four-layer TCP/IP model described in RFC 1122 (1989): link, internet, transport, application.
3. This chapter is entirely layer 1 and layer 2 in OSI terms, and entirely the link layer in TCP/IP terms.
4. The four link-layer duties have formal names: physical signalling, delineation, frame check sequence, and medium access control.
5. **Medium access control** (MAC) is the sublayer that decides who transmits when. Its name is also why hardware addresses are called MAC addresses.
6. Access methods in use: CSMA/CD (classic Ethernet), CSMA/CA (802.11 WiFi), token passing (802.5 Token Ring, FDDI), TDMA and OFDMA (cellular, WiFi 6).
7. The topology table below gives the practical trade-off.

Topology	Failure of one link	Where used now
Bus	Whole segment dies	Obsolete, CAN in cars
Star	One node dies	All LANs, home WiFi
Ring	Ring wraps, survives	SONET, metro fibre
Mesh	Reroute around it	Internet core, WiFi mesh

8. A **collision domain** is the set of ports where two simultaneous transmissions would interfere. A **broadcast domain** is the set of ports a broadcast frame reaches. Section 23.8 pulls these apart properly.
9. The distinction between a standard and a convention matters here. IEEE 802.3 is a standard. Calling the central box a "switch" rather than a "bridge" is a convention; the standard still says bridge.

23.1.6 WORDS - remember these

1. Medium — the stuff a message travels through — the physical channel: copper, fibre or free space.
2. Point-to-point — a private line between two machines — a link with exactly two endpoints and no contention.

3. Broadcast medium — a shared space everyone hears — a multiple-access channel requiring a MAC protocol.
4. Framing — marking where a message starts and stops — delineation of a bit stream into protocol data units.
5. Topology — the shape of the wiring — the graph of nodes and links, physical or logical.
6. Collision — two machines talking at once — overlapping transmissions on a shared medium producing an undecodable signal.
7. Layer — one job in the stack — an abstraction level with a defined service interface to the layer above.

23.2 Bits on a wire, physically

23.2.1 PLAIN - in simple words

1. A wire cannot carry a number. It can only carry a voltage, which is an electrical push.
2. So we agree on a code. High voltage means 1. Low voltage means 0. That is the simplest possible scheme.
3. This simple scheme has a name: **NRZ**, short for non-return-to-zero. The voltage just stays where it is for the whole bit.
4. NRZ has a serious problem, and the problem is time.
5. The receiver has to know exactly when each bit starts and ends. Otherwise it cannot tell one long 1 from four short 1s.
6. It could use its own clock, but no two clocks in the world run at exactly the same speed. They drift apart.
7. So the receiver needs the signal itself to tell it the time. It does this by watching for the moments the voltage changes.
8. Every change is a tick it can use to correct its clock. This is called **clock recovery**.
9. Send a hundred zeros in a row and there are no changes for a hundred bit times. The receiver's clock drifts, and it loses count. That is the failure.
10. There is a second problem. A long run of the same voltage builds up a steady electrical offset in the transformers and capacitors along the path.
11. That offset is called **DC bias**, and it slowly ruins the receiver's idea of what counts as high and what counts as low.
12. So we need a code where the voltage keeps changing no matter what the data says, and where highs and lows come out roughly equal over time. That balance is called **DC balance**.

23.2.2 PLAIN - a picture in your head

1. Think of a drummer keeping time for a dancer who cannot see the drummer.
2. The dancer knows the tempo roughly, but not perfectly. Over a long silence, the dancer slips.
3. If the drummer hits the drum on every single beat, the dancer never slips, because every beat is confirmed.
4. Plain NRZ is a drummer who only hits the drum when the music changes. During a long steady passage there is nothing to hear.
5. Manchester coding is a drummer who hits the drum in the middle of every single beat, always, and uses the direction of the hit to carry the message.
6. It costs twice the effort, but the dancer never loses the beat.
7. The clever later codes are drummers who found a way to guarantee a hit at least every few beats, without hitting on every beat.

Where this comparison breaks: a dancer who slips can recover by watching other dancers. A receiver has nothing else to watch. Also, a drum hit is either there or not, while a real signal on a long cable arrives smeared, rounded and mixed with echoes of earlier bits. The receiver is not detecting a clean edge, it is running an equalizer that reconstructs a plausible edge from a blurred one. At 10 Gbit/s over 100 m of

copper, the raw received waveform looks nothing like the transmitted one until that equalizer has done its work.

23.2.3 PLAIN - a worked example

1. We will encode one byte, the ASCII letter `K`, which is hex `0x4B` and binary `01001011`.
2. We use the IEEE 802.3 Manchester convention: a 0 is high then low, a 1 is low then high. Each bit becomes two half-bits.
3. Below, - is high voltage and _ is low. Each bit gets four characters, so two per half-bit.

```
bit    0   1   0   0   1   0   1   1
NRZ  ____----________----____--------
MAN  --____----__--____----____--__--
```

4. Look at the NRZ line. Bits 3 and 4 are both 0, so eight characters pass with no change at all. Bits 7 and 8 are both 1, same problem.
5. Now look at the Manchester line. There is a transition in the middle of every single bit, guaranteed, whatever the data is.
6. Count the highs and lows in the Manchester line. There are sixteen of each. Perfectly balanced. That is DC balance, guaranteed by construction.
7. The price is visible too. Manchester needs 32 half-bits to send 8 bits. It uses twice the signal changes per bit, so twice the bandwidth.
8. That is exactly why 10BASE-T, which used Manchester, needed 20 MHz of cable bandwidth to carry 10 Mbit/s.

23.2.4 PLAIN - what is really happening inside

1. A twisted pair is two copper wires twisted around each other along the whole length of the cable.
2. The transmitter does not put a voltage on one wire and use earth as the other. It drives the two wires in opposite directions.
3. If one goes up by one volt, the other goes down by one volt. The receiver only looks at the difference between them. This is **differential signalling**.
4. Now suppose a motor nearby throws out electrical noise. That noise hits both wires almost equally, pushing both up by the same amount.
5. The difference between them is unchanged. The noise cancels. Noise that affects both wires equally is called **common mode**, and differential receivers ignore it.
6. The twisting is what makes the noise hit both wires equally. Over each half twist, the wire nearer the noise source swaps places with the other one.
7. Averaged over the length, both wires have been equally near the noise. So the induced noise on the two is equal, and cancels.
8. Twisting also stops the pairs inside one cable from disturbing each other. That interference has a name, **crosstalk**, and each pair is twisted at a different rate so their fields do not line up.
9. Higher cable categories are, in large part, tighter and more precisely varied twist rates, better copper, and sometimes metal foil around each pair.
10. Foil or braid around the pairs is called **shielding**. It blocks outside interference, but only works if the shield is properly grounded at the ends.

23.2.5 TECHNICAL - the engineer's version

1. **NRZ** maps one bit to one symbol with two voltage levels. Zero coding overhead, no guaranteed transitions, no DC balance.
2. **Manchester** (IEEE 802.3 convention: 0 is high-to-low at mid-bit, 1 is low-to-high) is 50 percent efficient. It doubles the baud rate. Used by 10BASE-T, and still used by consumer infrared, RFID and NFC.
3. There are two Manchester conventions. G. E. Thomas defined the opposite mapping in 1949; inverting a Manchester signal converts one convention into the other. Always state which one you mean.

4. **4B/5B** maps each 4 data bits to a 5-bit symbol chosen so no symbol has more than three consecutive zeros. 80 percent efficient. Used by FDDI and by 100BASE-TX, where it feeds an MLT-3 line driver.
5. **MLT-3** is a three-level code that cycles low, mid, high, mid on each 1 and holds on each 0. It cuts the fundamental frequency of 100BASE-TX from 125 MHz to about 31.25 MHz, which is why Cat5 could carry Fast Ethernet.
6. **8b/10b** was described by Al Widmer and Peter Franaszek at IBM in 1983, US patent 4,486,739 granted December 1984. It maps 8 bits to 10, 80 percent efficient, guarantees run length and DC balance through a running disparity of at most plus or minus two, and supplies comma symbols for alignment. Used by 1000BASE-X, PCI Express 1.x and 2.x, SATA, USB 3.0, DisplayPort 1.x and Fibre Channel.
7. **64b/66b** arrived with 10 Gigabit Ethernet. It adds a 2-bit sync header to 64 bits of scrambled payload. Overhead is about 3.1 percent instead of 20 percent. It does not guarantee DC balance or run length; it makes them statistically overwhelmingly likely via a self-synchronous scrambler.
8. **PAM-4** sends two bits per symbol using four voltage levels instead of two. It halves the symbol rate for a given bit rate, at the cost of about 9.5 dB of signal-to-noise margin, so it is always paired with forward error correction. It is the basis of 50G, 100G, 200G and 400G electrical lanes.
9. **PAM-16**, sixteen levels, with 64B/65B framing and LDPC coding, is what 10GBASE-T uses over copper.

Code	Bits per symbol	Overhead	Example use
NRZ	1	0 percent	Short backplanes
Manchester	0.5	100 percent	10BASE-T
4B/5B + MLT-3	0.8	25 percent	100BASE-TX
8b/10b	0.8	25 percent	1000BASE-X
64b/66b	0.97	3.1 percent	10GBASE-R
PAM-4	2	needs FEC	400G lanes

10. Twisted-pair cabling categories are defined by ANSI/TIA-568 in North America and ISO/IEC 11801 internationally. The numbers below are the tested frequency, not the data rate.

Category	Bandwidth	Ethernet at 100 m
Cat 3	16 MHz	10BASE-T
Cat 5	100 MHz	100BASE-TX
Cat 5e	100 MHz	1000BASE-T
Cat 6	250 MHz	5GBASE-T
Cat 6A	500 MHz	10GBASE-T
Cat 7	600 MHz	10GBASE-T
Cat 7A	1000 MHz	10GBASE-T
Cat 8	2000 MHz	40GBASE-T at 30 m

11. Cat 5 was deprecated in favour of Cat 5e in 2001. Cat 7 and Cat 7A are ISO/IEC classes and were never adopted by TIA, and they use non-RJ45 connectors such as GG45 and TERA; this is why they are rare in practice.
12. Cat 8 comes in Class I (Cat 8.1, U/FTP or F/UTP, RJ45 compatible) and Class II (Cat 8.2, F/FTP or S/FTP, GG45 or TERA). It supports 25GBASE-T and 40GBASE-T, standardized in IEEE 802.3bq in 2016, over a maximum channel of 30 m. It is a data centre top-of-rack cable, not a building cable.
13. Shielding notation is `X/YTP` where X is the overall shield and Y is the per pair shield: UTP, F/UTP (foil overall), S/FTP (braid overall, foil per pair), U/FTP (foil per pair only).
14. Every twisted-pair Ethernet variant from 10BASE-T through 10GBASE-T is specified to 100 m of channel, which is 90 m of solid horizontal cable plus 10 m of stranded patch cords.

15. The honest version: category ratings are measured on a fully certified channel with proper terminations. A Cat 6A cable punched down badly, or run next to a fluorescent ballast, may fail at 30 m. Cable certification testers such as the Fluke DSX series measure insertion loss, NEXT, return loss and delay skew, and they exist precisely because the label on the jacket is a claim, not a measurement.

23.2.6 WORDS - remember these

1. NRZ — voltage just stays high or low — non-return-to-zero line code, one bit per symbol, no guaranteed transitions.
2. Clock recovery — working out the beat from the signal — extracting a bit clock from data transitions with a phase-locked loop.
3. DC balance — equal amounts of high and low — zero mean of the transmitted waveform, required for transformer and capacitor coupling.
4. Manchester — a change in the middle of every bit — biphase-L code, 50 percent efficient, self-clocking and DC balanced.
5. Line code — the rule turning bits into signals — the mapping from data bits to channel symbols.
6. Differential signalling — drive two wires oppositely — balanced transmission where the receiver takes the difference and rejects common-mode noise.
7. Crosstalk — one pair leaking into another — near-end and far-end coupling between pairs, measured as NEXT and FEXT in dB.
8. Twisted pair — two wires wound together — balanced copper pair with a controlled twist rate for noise and crosstalk rejection.
9. PAM-4 — four voltage levels instead of two — pulse amplitude modulation carrying two bits per symbol, always used with FEC.

23.3 Bits on fibre and on air

23.3.1 PLAIN - in simple words

1. Copper carries electricity. Fibre carries light. Air carries radio waves.
2. A fibre optic cable is a very thin strand of extremely pure glass. Thinner than a human hair.
3. A laser or an LED at one end flashes light into it. A light detector at the other end sees the flashes.
4. Light on means 1, light off means 0, in the simplest fibre systems.
5. The light does not leak out of the sides, even when the fibre is bent, and that is the whole trick.
6. It stays in because of **total internal reflection**. Light hitting the edge of the glass at a shallow enough angle bounces back in instead of escaping.
7. To make that happen, the fibre has two layers of glass. An inner **core** and an outer **cladding** with a slightly lower ability to bend light.
8. Fibre beats copper in three ways. It goes much further, it carries much more, and it does not care about electrical noise at all.
9. It has no metal in the signal path, so a lightning strike or a motor next door does nothing to it.
10. Radio is the third medium. It needs no cable at all, which is its whole appeal and also every one of its problems.

23.3.2 PLAIN - a picture in your head

1. Imagine a long straight corridor with mirrors on both walls.
2. Shine a torch down it at a shallow angle. The beam bounces from wall to wall and travels the whole length, still bright.
3. Now imagine the corridor is not straight but gently curved. The beam still bounces along, because the angles are still shallow.
4. Bend it sharply round a corner and some light hits the wall too steeply, passes through, and is lost. That is a **bend loss** in a real fibre.

5. Now imagine two kinds of corridor. A wide one, where beams entering at different angles take noticeably different path lengths.
6. Beams that started together arrive at different times, so a sharp flash at one end becomes a smeared blob at the other. That is **multi-mode** fibre.
7. And a very narrow corridor, so narrow that only one straight path fits. Every photon takes the same route and arrives together. That is **single-mode**.
8. Narrow costs more to build and needs a better torch. But the flash stays sharp over tens of kilometres.

Where this comparison breaks: light in a fibre is not a torch beam bouncing off mirrors. The refractive index change at the core boundary reflects the wave, and in single-mode fibre the core is only a few wavelengths wide, so ray optics stops being the right description entirely. There the light is a guided wave whose field extends slightly into the cladding. The corridor picture also has no equivalent of chromatic dispersion, where different colours of light travel at different speeds through the same glass and smear the pulse even in single-mode fibre.

23.3.3 PLAIN – a worked example

1. Compare three ways of getting 10 Gbit/s from one building to another.
2. Option one: 10GBASE-T over Cat 6A copper. Maximum 100 m. Cheap cable, cheap sockets, but the transceiver chip runs hot and burns several watts per port.
3. Option two: 10GBASE-SR over multi-mode fibre at 850 nm. Maximum 300 m on OM3 fibre, 400 m on OM4. Cheap laser, more expensive fibre.
4. Option three: 10GBASE-LR over single-mode fibre at 1310 nm. Maximum 10 km. More expensive laser, cheapest fibre per metre.
5. The buildings are 600 m apart. Copper is out at 100 m. Multi-mode is out at 400 m. Single-mode wins, and would still win at 6 km.
6. Notice that the fibre itself is not what sets the distance. The **optics** at each end do: the laser wavelength, its power, and the receiver's sensitivity.
7. Those optics live in a small plug-in module called an **SFP**. The switch port is a slot. You choose the distance by choosing the module.

Module type	Typical rate	Reach
SFP	1 Gbit/s	550 m to 80 km
SFP+	10 Gbit/s	300 m to 80 km
SFP28	25 Gbit/s	100 m to 10 km
QSFP28	100 Gbit/s	100 m to 40 km
QSFP-DD	400 Gbit/s	500 m to 40 km

23.3.4 PLAIN – what is really happening inside

1. The transmitter is a laser diode, or for the cheapest short links an LED. It is switched on and off billions of times a second.
2. The receiver is a photodiode. Light landing on it frees electrons and makes a tiny current, which an amplifier turns back into a digital signal.
3. Three things degrade the signal along the way, and they are different from the copper problems.
4. **Attenuation** is the light simply getting dimmer, absorbed and scattered by the glass. Measured in decibels per kilometre.
5. **Modal dispersion** is the smearing caused by different paths in multi-mode fibre. It is the reason multi-mode has a short reach.
6. **Chromatic dispersion** is the smearing caused by different wavelengths in the pulse travelling at slightly different speeds. It limits long single-mode links.

7. Glass is not equally transparent at all colours. There are three windows where it is unusually clear, and every optical network uses them.
8. Those windows are around 850 nanometres, 1310 nanometres and 1550 nanometres. All are infrared. None are visible to the eye.
9. 1550 nm is the clearest of the three, which is why undersea cables use it.
10. Because different wavelengths do not interfere, you can send many separate channels down one fibre at once, each on its own colour. That is **wavelength division multiplexing**.
11. Radio works on the same idea of a carrier wave, but the medium is shared with everyone in range, and the signal fades, reflects and gets absorbed by walls and bodies.

23.3.5 TECHNICAL - the engineer's version

1. Standard telecom fibre has a 125 micrometre cladding diameter regardless of type. Only the core differs.
2. Single-mode fibre has a core of about 8 to 10 micrometres. Multi-mode has 50 or 62.5 micrometres.
3. Single-mode is classified OS1 (indoor, tight buffered) and OS2 (outdoor, loose tube, lower loss). Multi-mode is OM1 through OM5.
4. Typical attenuation figures: about 3 dB/km at 850 nm on multi-mode, about 0.35 dB/km at 1310 nm and about 0.2 dB/km at 1550 nm on single-mode. Corning's Vascade EX2500 submarine fibre is specified at a nominal 0.148 dB/km at 1550 nm.
5. Total internal reflection occurs when light in the core meets the cladding at an angle greater than the critical angle, set by the two refractive indices. The **numerical aperture** is the sine of the largest input angle that will be guided; typical multi-mode NA is 0.2 to 0.29.
6. Single-mode fibre is used for essentially all links longer than about 1 km, and multi-mode inside buildings and data halls.

Fibre grade	Core	10G reach	100G reach
OM3	50 um	300 m	70 m
OM4	50 um	400 m	100 m
OM5	50 um	400 m	150 m
OS2	9 um	10 to 80 km	10 to 80 km

7. The pluggable optic form factors are defined by multi-source agreements (MSAs) between vendors, not by IEEE. SFP dates from 2001, SFP+ from 2006, QSFP+ from 2009, QSFP28 from 2014, QSFP-DD and OSFP from 2016 onward.
8. This is a good example of a **convention** rather than a standard: an MSA is an industry agreement with no standards body behind it, which is why vendor lock-in on optics is so common.
9. Coarse WDM (CWDM) uses 8 to 18 channels spaced 20 nm apart. Dense WDM (DWDM) uses 40 to 96 or more channels spaced 0.8 nm or 0.4 nm apart on the ITU-T G.694.1 grid. A single modern DWDM fibre pair carries tens of terabits per second.
10. Light travels through glass at about 200,000 km/s, roughly two-thirds of its speed in vacuum, because the refractive index of silica is about 1.47. That gives about 5 microseconds of delay per kilometre of fibre, and it is a hard physical floor on latency.
11. Radio is IEEE 802.11 for WiFi and 3GPP specifications for cellular. Chapter 31 covers radio fully: modulation, channels, the 2.4, 5 and 6 GHz bands, CSMA/CA, MIMO, OFDMA and why WiFi behaves so differently from a cable. We say only this much here: the reader's laptop reached the router over radio, and the fault in this book's running example was not in that radio hop.
12. Observation commands: on a switch, `show interfaces transceiver` reports per-lane optical transmit and receive power in dBm, which is how you tell a dirty connector from a dead laser.

23.3.6 WORDS - remember these

1. Core — the middle of the fibre that carries light — the higher-refractive index region, 8 to 10 um single-mode, 50 or 62.5 um multi-mode.
2. Cladding — the outer glass that keeps light in — the lower-index layer, standard 125 um outer diameter.
3. Total internal reflection — light bouncing back instead of escaping — reflection above the critical angle at a refractive-index boundary.
4. Single-mode — one path only, goes far — small-core fibre supporting one propagation mode, no modal dispersion.
5. Multi-mode — many paths, goes less far — large-core fibre with modal dispersion limiting reach and bandwidth.
6. Attenuation — the signal getting weaker — optical power loss in dB per km.
7. Dispersion — the pulse smearing out — modal or chromatic pulse spreading that limits the bit rate over distance.
8. SFP — the plug-in module that does the light — small form-factor pluggable transceiver defined by an MSA, not by IEEE.
9. WDM — many colours down one fibre — wavelength division multiplexing on the ITU-T G.694.1 grid.

23.4 Bandwidth, throughput, latency and jitter

23.4.1 PLAIN - in simple words

1. People say “my internet is slow” and mean four completely different things.
2. **Bandwidth** is the maximum rate the link can carry. It is a capacity. It is what you bought.
3. **Throughput** is the rate you actually get for a particular transfer, right now. It is always less than bandwidth, sometimes far less.
4. **Latency** is the waiting time before anything arrives. How long one message takes to get there.
5. **Jitter** is how much that waiting time wobbles from message to message.
6. A video call needs low latency and low jitter. It barely needs bandwidth.
7. A large download needs bandwidth. It does not care about latency at all.
8. A game needs low latency above everything.
9. These four are almost independent. A link can have enormous bandwidth and terrible latency, and a link can have tiny bandwidth and superb latency.
10. And there is a fifth state that is none of these: the link can be silent. Nothing comes back at all. That is what happened to the reader.

23.4.2 PLAIN - a picture in your head

1. The classic picture is a water pipe.
2. Bandwidth is how wide the pipe is. A wide pipe moves more water per second.
3. Latency is how long the pipe is. A long pipe means water takes longer to appear at the far end, even if the pipe is very wide.
4. Throughput is how much water actually comes out per second, which depends on the pipe, the tap, the pump, and anything half-blocking the middle.
5. Jitter is the pulsing you feel when the flow is not steady.
6. Widening the pipe does nothing for the time the first drop takes to appear. That is the single most useful thing this picture teaches.

Where this comparison breaks: it breaks in three important places, and engineers get caught by all three. First, water in a pipe is continuous, but data is sent in discrete packets, so a packet that is one byte too big for the pipe is not trimmed, it is dropped whole. Second, a water pipe does not lose water and then resend it, but a network does, and the resend costs a full round trip of latency that the pipe picture cannot represent. Third, and most importantly here, a blocked water pipe still tells you something: pressure

builds, or water comes back. A silently dropping network gives you nothing at all. There is no equivalent of a packet vanishing without trace in the water picture, and that vanishing is precisely what the reader was looking at.

23.4.3 PLAIN - a worked example

1. The reader's own case is the clearest example in this book.
2. The command was `curl -v https://github.com`. It printed `Trying 20.207.73.82:443...` and then nothing at all for fifteen seconds.
3. Consider what each of the four measures would have looked like if it were the problem.
4. Low bandwidth: the page would have loaded, slowly, byte by byte. It did not load at all.
5. High latency: the page would have loaded after a long pause. It never loaded.
6. Low throughput because of congestion: the transfer would have stalled and restarted. There was no transfer to stall.
7. High jitter: the page would have loaded unevenly. There was nothing to be uneven about.
8. What actually happened is that the connection attempt got no reply of any kind. No refusal, no error, no data. Silence.
9. That is not slowness. That is a packet going into the network and never producing any observable effect.
10. The same site loaded instantly on mobile data from the same phone. So the site was up, and the destination was reachable from India.
11. What that proves: the failure was specific to the path taken from the reader's broadband line, not to the destination.
12. What it merely suggests: something on that path was discarding packets silently. It does not tell us which device, or why. Later chapters in Part F narrow that down.

23.4.4 PLAIN - what is really happening inside

1. Latency is made of four separate delays, and they add up.
2. **Propagation delay** is the time the signal takes to physically travel. In fibre, about 5 microseconds per kilometre. Nothing can beat this.
3. **Transmission delay** is the time to push all the bits of a message out of the interface. It equals message size divided by link rate.
4. **Processing delay** is the time each router takes to look at the packet and decide where to send it. Modern hardware does this in microseconds.
5. **Queuing delay** is the time the packet spends waiting in a buffer behind other packets. This is the one that varies, and it is the main source of jitter.
6. Now the key idea. A link with high bandwidth and high latency can hold a surprising amount of data that has left the sender but not yet arrived.
7. That quantity has a name: the **bandwidth-delay product**. It is the capacity of the pipe itself, measured in bytes in flight.
8. If the sender is only allowed a certain number of unacknowledged bytes at a time, and that number is smaller than the bandwidth-delay product, the link can never be filled, however fast it is.
9. This is why a 1 Gbit/s link from India to the United States can deliver 3 Mbit/s to a badly tuned application. The link is not slow. The window is too small.

23.4.5 TECHNICAL - the engineer's version

1. Bandwidth in networking means capacity in bits per second. In signal processing it means a frequency range in hertz. The two are related through the Shannon-Hartley theorem but are not the same quantity. Say which you mean.
2. Round-trip time (RTT) is what `ping` measures. One-way latency is roughly half of it, but only on a symmetric path, and many paths are not symmetric.
3. Bandwidth-delay product, worked in full:

```
Link rate      = 100 Mbit/s = 100,000,000 bits/s
RTT            = 180 ms     = 0.180 s
BDP (bits)     = 100e6 * 0.180 = 18,000,000 bits
BDP (bytes)    = 18,000,000 / 8 = 2,250,000 bytes
               = about 2.15 MiB in flight
```

4. 180 ms is a realistic round trip from an Indian home line to a server on the US east coast. 2.15 MiB must be in flight at all times to keep that link full.
5. The original TCP receive window field is 16 bits, so a maximum of 65,535 bytes. That is 3 percent of what this link needs.
6. RFC 1323 (1992), now superseded by RFC 7323 (2014), added the window scale option, which multiplies the window by a power of two up to 2^14. Without it, long fat networks cannot be filled.
7. Maximum achievable throughput with a fixed window is `window / RTT`. With 65,535 bytes and 180 ms that is about 2.9 Mbit/s, on a 100 Mbit/s link.
8. TCP and its congestion control get a full chapter later in Part F. The point here is only that link speed alone never predicts throughput.

Quantity	Unit	Typical home value
Bandwidth	Mbit/s	100 to 1000
Throughput	Mbit/s	40 to 900
Latency to gateway	ms	0.5 to 3
Latency India to US	ms	180 to 280
Jitter, good WiFi	ms	1 to 5
Jitter, congested	ms	30 to 200

9. Jitter is formally the variation in one-way delay. RFC 3550, the RTP specification (2003), defines an interarrival jitter estimator that media applications use to size their playout buffer.
10. **Bufferbloat** is the condition where oversized, unmanaged buffers in a modem or router hold hundreds of milliseconds of packets, so latency explodes under load while bandwidth tests still look fine. Jim Gettys named and characterized it in 2010. The fixes are active queue management algorithms: CoDel (2012) and fq_codel, and the CAKE qdisc on Linux.
11. Tools: `ping` for RTT, `iperf3` for throughput, `mtr` for per-hop loss and latency, and `flent` running the RRUL test for bufferbloat under load.
12. The honest version: `ping` measuring 20 ms does not mean your data has 20 ms of latency. ICMP echo is often handled on a different, slower or faster path inside a router than forwarded traffic, and is frequently rate-limited or deprioritized. This matters enormously for reading a traceroute, which is a later chapter in Part F.
13. For the reader's case none of these numbers apply, because there was no measurable transfer. The correct technical description is a connection attempt that received neither a SYN-ACK, nor a TCP RST, nor an ICMP unreachable message, within the 15 second observation window.

23.4.6 WORDS - remember these

1. Bandwidth — how wide the pipe is — link capacity in bits per second, or a frequency range in hertz depending on context.
2. Throughput — what you actually get — achieved data rate for a given flow over a given interval, also called goodput when headers are excluded.
3. Latency — the waiting time before anything happens — one-way or round-trip delay, the sum of propagation, transmission, processing and queuing delays.
4. Jitter — how much the wait wobbles — variance in one-way delay, estimated per RFC 3550 in real-time media.

5. Propagation delay — the time light or electricity takes to travel — distance divided by signal velocity, about 5 us per km in fibre.
6. Bandwidth-delay product — how much data fits inside the link — link rate multiplied by RTT, the in-flight capacity in bytes.
7. Bufferbloat — big buffers making everything laggy — excess queuing delay from unmanaged FIFO buffers, fixed by AQM such as fq_codel.
8. Silent drop — nothing comes back at all — a discard with no ICMP or TCP notification, indistinguishable from a black hole from the sender's side.

23.5 Ethernet

23.5.1 PLAIN - in simple words

1. Ethernet is the set of rules for sending data over a local network cable. It is more than fifty years old and it won completely.
2. It was invented in 1973 at a research lab in California called Xerox PARC, by Robert Metcalfe and David Boggs.
3. The idea it copied came from Hawaii. A radio network called ALOHAnet let many machines share one radio channel by simply transmitting and retrying after a clash.
4. Metcalfe's version used a cable instead of radio, and added one crucial rule: listen before you talk.
5. The name comes from "luminiferous ether", the invisible substance nineteenth century physicists thought light travelled through. It was a joke about the cable being the medium everything swims in.
6. In the beginning, every machine was tapped onto one thick coaxial cable running through the building. One cable, shared by all.
7. Because it was shared, two machines could talk at once and ruin each other's message. That is a **collision**.
8. The rule for handling it was: listen first, and if a collision happens anyway, stop, wait a random time, and try again.
9. That rule is CSMA/CD, which stands for carrier sense multiple access with collision detection.
10. Today, on almost every network you will ever touch, collisions cannot happen, because the shared cable is gone.

23.5.2 PLAIN - a picture in your head

1. Picture a small meeting where everyone shares one microphone that is always on.
2. Before speaking, you listen. If someone is talking, you wait. That is carrier sense.
3. Sometimes two people start in the same instant, because each heard silence. Both voices go out together and neither is understandable. That is a collision.
4. Both notice the clash while it is happening, stop immediately, and each waits a different random number of seconds before trying again. That is collision detection and backoff.
5. Now replace the meeting with a room where every person has a private two-way headset to a chairperson.
6. Anyone can speak at any time, in both directions at once, and the chairperson passes each message only to the person it is for.
7. Nobody ever clashes with anybody. That is a modern switched, full-duplex network.

Where this comparison breaks: humans hear a clash instantly. On a long cable, electricity takes time to travel, so a station at one end can be halfway through transmitting before it hears the clash from the far end. This is why classic Ethernet had a minimum frame size and a maximum cable length that were mathematically linked. If a frame were too short, the sender could finish before the collision signal got back, and would never know it had failed.

23.5.3 PLAIN – a worked example

1. Take the original 10 Mbit/s shared Ethernet and work out why the minimum frame is 64 bytes.
2. The rule is: a sender must still be transmitting when a collision from the far end reaches it.
3. Worst case, the signal must go all the way to the far end and the collision must come all the way back. That is a full round trip.
4. The standard fixes this round trip budget at **512 bit times**, which is the time to send 512 bits.

```
512 bits at 10 Mbit/s = 51.2 microseconds
512 bits / 8          = 64 bytes
```

5. So a frame must be at least 64 bytes long, or the sender might finish too early. 64 bytes counts from the destination address through the check sequence.
6. That 51.2 microsecond budget also caps the cable. Signals travel through coax at roughly two thirds the speed of light, and repeaters add delay, so the maximum end-to-end network was 2500 m with at most four repeaters.
7. If a frame arrives shorter than 64 bytes, it is called a **runt** and is discarded. Historically a runt meant a collision. Today it means a faulty cable, a bad NIC or a duplex mismatch.
8. On a switched full-duplex link there is no shared medium and no collision, so none of this timing matters. The 64-byte minimum survives anyway, purely for compatibility.

23.5.4 PLAIN – what is really happening inside

1. A modern Ethernet link between a laptop and a switch is two independent one-way channels using different wire pairs.
2. Each end can transmit whenever it likes, because nobody else is on its transmit path. This is **full duplex**.
3. When the cable is plugged in, the two ends negotiate. They exchange short pulse bursts advertising every speed and duplex mode they support.
4. They then both pick the best mode that both support. This is **auto-negotiation**, and it is why plugging in a cable just works.
5. If negotiation fails or is disabled at one end, you can get a **duplex mismatch**: one side full duplex, the other half duplex.
6. That link will still pass traffic, but the half-duplex side will report collisions and late collisions and throughput will collapse under load. It is a classic slow-network fault.
7. The switch in the middle looks at the destination address of each frame and sends it out one port only. Section 23.8 explains exactly how it knows which.
8. Because each link is private and full duplex, the network has as many collision domains as it has ports, and each of those domains has exactly one possible talker in each direction. So there are no collisions.

23.5.5 TECHNICAL – the engineer's version

1. The experimental Ethernet first ran on 22 May 1973 at Xerox PARC at 2.94 Mbit/s. That odd rate was the Alto computer's system clock divided down.
2. Metcalfe and Boggs published "Ethernet: Distributed Packet Switching for Local Computer Networks" in Communications of the ACM in July 1976.
3. US patent 4,063,220, "Multipoint data communication system with collision detection", was granted in 1977 to Xerox, naming Metcalfe, Boggs, Thacker and Lampson.
4. The DIX consortium of Digital Equipment Corporation, Intel and Xerox published the Ethernet Blue Book in 1980 at 10 Mbit/s. Version 2, the frame format still called Ethernet II, was published in November 1982.
5. Metcalfe founded 3Com in June 1979 to commercialize it.
6. IEEE published 802.3 as a draft in 1983 and as a full standard in 1985. The competing standards, Token Ring (IEEE 802.5, 1985) and FDDI (ANSI X3T9.5, 1987), were technically respectable and lost on cost.
7. The 802.3 speed family, with the amendment letter and year:

Amendment	Year	What it added
802.3	1983	10BASE5, 10 Mbit/s coax
802.3i	1990	10BASE-T, twisted pair
802.3u	1995	100BASE-TX, Fast Ethernet
802.3z	1998	1000BASE-X, gigabit fibre
802.3ab	1999	1000BASE-T, gigabit copper
802.3ae	2002	10 Gbit/s over fibre
802.3an	2006	10GBASE-T over copper
802.3ba	2010	40 and 100 Gbit/s
802.3bz	2016	2.5G and 5GBASE-T
802.3bq	2016	25G and 40GBASE-T
802.3bs	2017	200 and 400 Gbit/s
802.3df	2024	800 Gbit/s, 100G lanes

8. Media naming is systematic: the number is the speed in Mbit/s or Gbit/s, BASE means baseband (no carrier modulation of the whole channel), and the suffix names the medium. T is twisted pair, S is short-wave multi-mode fibre, L is long-wave single-mode, C is twinaxial copper, R is a 64b/66b coded lane.
9. 10BASE5 was 10 mm thick coax, up to 500 m per segment, tapped with a vampire tap that pierced the jacket. 10BASE2 was thinner coax with BNC T-pieces, 185 m per segment, and was universally called thinnet or cheapernet.
10. The connector story: coax with N-type or BNC, then the 8P8C modular plug that everyone calls RJ45 for 10BASE-T onward. RJ45 is technically the wrong name, since RJ45 is a telephone registered jack specification, but the usage is universal. That is a convention, not a standard.
11. Pin assignments come from TIA-568A and TIA-568B, which differ only in swapping the green and orange pairs. A cable with one end A and one end B is a crossover cable.
12. Crossover cables are no longer needed because Auto-MDI-X, standardized as part of 802.3ab in 1999 and now universal, detects and swaps the pairs electronically.
13. Interframe gap is 96 bit times: 9.6 us at 10 Mbit/s, 0.96 us at 100 Mbit/s, 96 ns at 1 Gbit/s. It exists so receivers can resynchronize between frames.
14. Half-duplex gigabit exists in the standard, using carrier extension to pad the slot time to 4096 bit times, but no one ever deployed it.
15. Observation: `ethtool eth0` on Linux shows negotiated speed, duplex and auto-negotiation state; `networksetup -listallhardwareports` and `ifconfig` on macOS show the equivalent, and `netstat -i` shows error counters.

23.5.6 WORDS - remember these

1. Ethernet — the standard way to wire a local network — the IEEE 802.3 family of physical and MAC layer specifications.
2. CSMA/CD — listen, talk, stop if you clash — carrier sense multiple access with collision detection, mandatory on half-duplex shared media.
3. Collision — two stations transmitting at once — overlapping transmissions within one collision domain, detected by the transceiver.
4. Backoff — waiting a random time before retrying — truncated binary exponential backoff over a doubling contention window.
5. Full duplex — both directions at once — simultaneous independent transmit and receive on separate pairs or wavelengths.
6. Auto-negotiation — the cable sorting itself out — link code word exchange using fast link pulses to select speed, duplex and flow control.

7. Duplex mismatch — one side full, one side half — a misconfiguration causing late collisions and severe throughput loss.
8. Runt — a frame that is too short — a frame under 64 octets, discarded and counted as an error.
9. Interframe gap — the compulsory pause between frames — 96 bit times of idle before the next preamble.

23.6 The Ethernet frame, field by field

23.6.1 PLAIN - in simple words

1. Everything sent on an Ethernet network is wrapped in a **frame**.
2. A frame is like an envelope. The data you care about is the letter inside. The envelope carries the addressing and the checks.
3. The envelope has a fixed layout, and every network card in the world agrees on it. That agreement is what makes any card talk to any other.
4. It begins with a wake-up pattern so the receiver can lock onto the timing.
5. Then the address it is going to, then the address it came from.
6. Then a two-byte code saying what kind of thing is inside. Is this an IP packet, or an ARP message, or something else.
7. Then the data itself, between 46 and 1500 bytes.
8. Then a four-byte check number computed from everything before it.
9. The receiver recomputes that check number. If it does not match, the frame is thrown away silently. Ethernet does not ask for a resend. Something higher up must notice.
10. The whole frame, not counting the wake-up pattern, is between 64 and 1518 bytes.

23.6.2 PLAIN - a picture in your head

1. Think of a postcard going through an old sorting office.
2. The first thing the machine needs is a run of identical marks along the edge so it can line the card up and match its rollers to the card's speed. That is the preamble.
3. Then one different mark saying "the card starts here". That is the start frame delimiter.
4. Then the delivery address, then the return address.
5. Then a small printed code saying what class of item this is: a letter, a parcel slip, a legal notice. That is the EtherType.
6. Then the message.
7. Then a checksum stamped at the end, computed from the whole card by the sending office. The receiving office recomputes it and bins the card if they disagree.
8. Postcards have a minimum size because the machine cannot grip anything smaller, and a maximum size because the slot is only so wide.

Where this comparison breaks: a sorting office that bins a damaged card usually tells someone. Ethernet tells nobody. A frame failing its check sequence is counted in a statistics register on the interface and then forgotten. Also, a postcard's return address is written by the sender and can be a lie; so can an Ethernet source address, and that is the root of several attacks in section 23.10.

23.6.3 PLAIN - a worked example

1. Here is a real frame, an ARP request from the reader's laptop asking who owns `192.168.0.1`. MAC addresses are illustrative, since the session record does not include them.

```
Preamble  AA AA AA AA AA AA AA      (7 bytes, alternating)
SFD       AB                        (1 byte)
Dest MAC  FF FF FF FF FF FF         (6 bytes, broadcast)
Src MAC   3C 22 FB 8A 41 D7         (6 bytes, the laptop)
EtherType 08 06                     (2 bytes, ARP)
```

```
Payload   00 01 08 00 06 04 00 01   (28 bytes of ARP,
          3C 22 FB 8A 41 D7          padded to 46)
          C0 A8 00 6B
          00 00 00 00 00 00
          C0 A8 00 01
Padding   00 x 18                   (to reach 46 bytes)
FCS       9C 4E 1B 33               (4 bytes, CRC-32)
```

2. `C0 A8 00 6B` is 192.168.0.107 in hexadecimal. That is the laptop.
3. `C0 A8 00 01` is 192.168.0.1 in hexadecimal. That is the reader's router.
4. Count the frame from destination MAC to FCS: 6 + 6 + 2 + 46 + 4 = 64 bytes. Exactly the minimum.
5. The ARP message itself is only 28 bytes, so 18 bytes of zero padding were added to reach the 46-byte minimum payload.
6. The preamble and SFD are not counted in the 64. They are physical-layer framing, stripped before the frame is handed up.
7. A packet capture tool will normally show you this frame as 60 bytes, because it also strips the 4-byte FCS after checking it.

23.6.4 PLAIN - what is really happening inside

1. The preamble is 56 bits of alternating 1 and 0. On the wire this is a square wave, which is the easiest possible thing for a receiver's clock circuit to lock onto.
2. The start frame delimiter breaks the pattern with two 1 bits in a row. That single break is what says "data starts on the next bit".
3. The EtherType field does double duty, and this is a historical scar. Values of 1500 or below mean "this is a length in bytes", the original IEEE 802.3 meaning. Values of 1536 or above mean "this is a type code", the DIX Ethernet II meaning.
4. Because 1500 is below 1536, both interpretations can coexist unambiguously, which is why the maximum payload is 1500 and not something rounder.
5. The check sequence is a CRC-32, a cyclic redundancy check. It is not a simple sum. The frame is treated as one enormous binary number and divided by a fixed polynomial; the remainder is the check value.
6. CRC-32 catches all single-bit errors, all double-bit errors, all odd numbers of errors, and all burst errors up to 32 bits. Beyond that it catches all but about 1 in 4 billion.
7. The maximum payload of 1500 bytes is called the **MTU**, maximum transmission unit.
8. The number 1500 came from the 1980 DIX specification. Buffer memory was expensive in 1979, and a station holding the shared coax cable for too long starved everybody else. 1500 bytes at 10 Mbit/s is about 1.2 milliseconds of airtime, which was judged a fair maximum turn.
9. It has never changed, on any Ethernet speed, for compatibility. A 400 Gbit/s link still defaults to a 1500-byte MTU chosen for a 10 Mbit/s coax cable in

1979.

10. **Jumbo frames** raise the payload to around 9000 bytes. Nine thousand was chosen because CRC-32's error detection strength starts to weaken beyond about 12000 bytes, and 9000 comfortably holds an 8192-byte block plus headers.
11. Jumbo frames are not in the IEEE standard. They are a vendor convention. Every device on the path must agree, or frames are silently dropped.

23.6.5 TECHNICAL - the engineer's version

1. The complete frame layout, in transmission order:

Field	Bytes	Purpose
Preamble	7	Clock synchronization

Field	Bytes	Purpose
Start frame delimiter	1	Marks start of frame
Destination MAC	6	Receiver address
Source MAC	6	Sender address
802.1Q tag	4	Optional VLAN tag
EtherType or length	2	Payload type or size
Payload	46 to 1500	The data
Frame check sequence	4	CRC-32
Interframe gap	12	Idle before next frame

2. Preamble is `10101010` repeated seven times. The SFD is `10101011`. Together they are 64 bits, often described as an 8-byte preamble ending in two 1 bits.
3. Minimum frame is 64 octets measured from destination MAC through FCS inclusive. Preamble, SFD and IFG are not counted.
4. Maximum untagged frame is 1518 octets: 6 + 6 + 2 + 1500 + 4. With one 802.1Q tag it is 1522. With two stacked tags (802.1ad, "QinQ") it is 1526.
5. With an 802.1Q tag present the minimum payload is 42 bytes rather than 46, so the 64-octet minimum still holds.
6. The FCS is a left-shifting CRC-32 with polynomial 0x04C11DB7, initial value 0xFFFFFFFF, post-complemented, giving a residue check value of 0x38FB2284.
7. Common EtherType values, all in hexadecimal, assigned by the IEEE Registration Authority:

EtherType	Protocol
0x0800	IPv4
0x0806	ARP
0x8100	802.1Q VLAN tag
0x86DD	IPv6
0x8847	MPLS unicast
0x88CC	LLDP
0x8863	PPPoE discovery
0x8864	PPPoE session

8. MTU is the layer 3 payload limit, 1500 bytes for standard Ethernet. It is distinct from the maximum frame size, which includes the 18 bytes of header and FCS.
9. Fragmentation costs, and this is why MTU matters. If an IP packet is larger than the path MTU, it must be split. Every fragment carries a full IP header, so overhead rises. If any one fragment is lost, the whole original packet is lost. Firewalls and load balancers often handle fragments badly or drop them. Fragment reassembly consumes memory at the receiver and is a classic denial of service target.
10. IPv4 fragmentation is defined in RFC 791 (1981). IPv6 removed router fragmentation entirely in RFC 8200 (2017); only the source may fragment.
11. Path MTU discovery, RFC 1191 (1990) for IPv4 and RFC 8201 (2017) for IPv6, finds the smallest MTU on a path by sending packets with the Don't Fragment bit set and listening for ICMP "fragmentation needed" replies.
12. The honest version: PMTUD is fragile in the real world, because many networks block all ICMP, so the "fragmentation needed" message never arrives and the connection hangs after the handshake. This is the classic PMTUD black hole, and RFC 4821 (2007) defines Packetization Layer PMTUD to work around it by probing at the TCP layer instead.
13. PPPoE, used on many DSL lines, adds 8 bytes of header and leaves an MTU of 1492. VPNs and tunnels reduce it further, which is why tunnel interfaces often default to 1400 or lower.

14. The reader's macOS machine showed several `utun` interfaces. Those are tunnel interfaces, and each carries its own smaller MTU. Check with `ifconfig utun0` and look for the `mtu` value.
15. Jumbo frame support is queried with `ip link show` on Linux or `ifconfig` on macOS; on a switch it is often called "system jumbo mtu" or "maximum frame size" and applies per port or globally depending on the vendor. That is an implementation detail, not a standard.

23.6.6 WORDS - remember these

1. Frame — the envelope around your data on a local network — a layer 2 protocol data unit delimited by preamble and interframe gap.
2. Preamble — the wake-up pattern at the start — 56 bits of alternating 1 and 0 for receiver clock recovery.
3. EtherType — the code saying what is inside — a 2-byte protocol identifier when the value is 1536 or greater.
4. Payload — the actual data being carried — 46 to 1500 octets in a standard untagged frame.
5. FCS — the check number at the end — a 32-bit CRC over the frame, polynomial 0x04C11DB7.
6. MTU — the largest chunk of data one frame can carry — maximum transmission unit, 1500 octets for standard Ethernet.
7. Jumbo frame — a bigger than normal frame — a non-standard payload of about 9000 octets, requiring agreement across the whole path.
8. Fragmentation — cutting a packet up to fit — splitting an IP datagram across multiple frames, with per-fragment header cost and all-or-nothing loss.
9. Runt and giant — too small and too big — frames below 64 or above the configured maximum, both counted as errors.

23.7 MAC addresses

23.7.1 PLAIN - in simple words

1. Every network interface has a hardware address burned into it. It is called a **MAC address**.
2. MAC stands for media access control. The name is an accident of which sublayer of the standard defines it.
3. It is 48 bits long, which is 6 bytes. It is written as six pairs of hexadecimal digits.
4. Example: `3C:22:FB:8A:41:D7`. Some systems use hyphens or dots instead of colons. Same address.
5. It is meant to be globally unique, so that no two interfaces anywhere have the same one.
6. The first three bytes identify the manufacturer. The last three are the manufacturer's own serial number for that interface.
7. So from a MAC address you can usually tell what kind of device it is, without asking the device.
8. An address is either for one device, for a group of devices, or for everyone on the network.
9. The everyone address is all ones: `FF:FF:FF:FF:FF:FF`. A frame sent there is delivered to every device on the local network.
10. Modern phones and laptops now often make up a fake MAC address on purpose, to stop shops and networks tracking you as you walk around.

23.7.2 PLAIN - a picture in your head

1. Think of a MAC address like the serial number stamped on the back of an appliance.
2. The first part identifies the factory. The second part identifies the individual unit that factory made.
3. Two appliances from the same factory share the first part and differ in the second.
4. Unlike a postal address, it says nothing about where the device is. It travels with the device.
5. A postal address changes when you move house. A serial number does not. This is exactly the difference between an IP address and a MAC address.
6. That difference is why both exist, and it is the reason ARP has to exist at all.

Where this comparison breaks: a serial number is permanent, but a MAC address can be changed in software on almost every operating system in about one command. Uniqueness is administrative, not physical. Manufacturers have shipped duplicate addresses by mistake, and cheap unbranded network chips have shipped whole batches with identical addresses.

23.7.3 PLAIN - a worked example

1. Take the address `3C:22:FB:8A:41:D7`.
2. Split it: `3C:22:FB` is the vendor part, `8A:41:D7` is the device part.
3. The vendor part is called the **OUI**, organizationally unique identifier. It is bought from the IEEE.
4. `3C:22:FB` is registered to Apple. So this is very likely a Mac or an iPhone.
5. You can look up any OUI in the IEEE public registry, or with a local tool. On many Linux systems, `/usr/share/ieee-data/oui.txt` holds the whole list.
6. Now look at the first byte alone: `3C` is `00111100` in binary.
7. The last bit of that first byte is 0. That means unicast: one device.
8. If it were 1, it would be multicast: a group. `01:00:5E:...` is the IPv4 multicast group prefix, and `33:33:...` is the IPv6 one.
9. The second-to-last bit of the first byte is 0. That means the address is globally assigned by a real manufacturer.
10. If it were 1, the address is **locally administered**, meaning somebody made it up. Randomized privacy addresses always have this bit set to 1.
11. So `3C` ends in `00`: globally unique, unicast, real Apple hardware.
12. Compare with `3E:22:FB:8A:41:D7`. `3E` is `00111110`. Last bit 0, so still unicast, but second-to-last bit is 1, so this one was made up locally.

23.7.4 PLAIN - what is really happening inside

1. The network card holds its address in a small memory chip, and the driver reads it when the interface starts.
2. When the card receives a frame, it compares the destination address against a short list without waking the CPU.
3. The list is: its own address, the broadcast address, and any multicast groups it has been told to join.
4. If the destination matches nothing on that list, the card discards the frame in hardware. The operating system never sees it.
5. Multicast matching is often done with a hash filter rather than exact matching, so the card sometimes lets through groups it did not join. The driver filters those out in software.
6. There is a special mode where the card stops filtering entirely and passes up everything it hears. That is **promiscuous mode**, and section 23.12 covers it.
7. Address randomization works by generating a random 46-bit value, setting the locally administered bit to 1 and the multicast bit to 0, and using that as the interface address.
8. On phones, a different random address is typically generated per network name, so the same shop sees a consistent device but two different shops cannot link you.

23.7.5 TECHNICAL - the engineer's version

1. The 48-bit format is IEEE EUI-48. There is also a 64-bit EUI-64 used by IPv6 interface identifiers, FireWire and ZigBee.
2. The IEEE Registration Authority sells three block sizes: MA-L (a 24-bit OUI, 16.7 million addresses), MA-M (28-bit prefix, 1 million addresses) and MA-S (36-bit prefix, 4096 addresses).
3. Bit ordering matters and confuses everyone. Ethernet transmits each octet least-significant bit first, so the I/G bit is literally the first bit on the wire, which is why the hardware can begin filtering immediately.
4. Bits in the first octet:

Bit	Name	Meaning if 1
Least significant	I/G	Multicast or broadcast
Second least	U/L	Locally administered

5. Broadcast is `FF:FF:FF:FF:FF:FF`, all 48 bits set, which is simply the all-groups multicast address.
6. IPv4 multicast maps to `01:00:5E` plus the low 23 bits of the group address, per RFC 1112 (1989). IPv6 multicast maps to `33:33` plus the low 32 bits, per RFC 2464 (1998).
7. Ranges reserved for local use without IEEE registration are defined in IEEE 802c (2017), which carves the locally administered space into structured quadrants.
8. MAC randomization history, with dates, since this changes fast: Apple iOS 8 randomized addresses during WiFi scanning in 2014; Android 10 made randomized addresses the default for connections in 2019; Apple shipped Private WiFi Address in iOS 14 and iPadOS 14 in 2020, and macOS gained a rotating variant later. Windows 10 has had a per-network random hardware address option since 2015.
9. Since iOS 18 and macOS 15 (2024) Apple's default is a rotating private address that changes periodically on networks the device does not consider trusted. Check current behaviour before relying on it; this area moves.
10. Consequences you must plan for: MAC-based DHCP reservations break, MAC filtering as a security control was always weak and is now useless, and network access control must use 802.1X with real credentials instead.
11. Commands to read your own address:

```
# macOS, all interfaces with hardware addresses
ifconfig | grep ether

# macOS, which port is which
networksetup -listallhardwareports

# macOS, the WiFi interface specifically
ifconfig en0 | grep ether

# Linux
ip link show
cat /sys/class/net/eth0/address
```

12. To see the vendor decoded for you, packet capture tools do it automatically: `tcpdump -e` prints Ethernet headers, and Wireshark shows `Apple_8a:41:d7` style names using its own OUI database.
13. The honest version: MAC addresses do not travel across routers. A router strips the incoming frame and builds a new one with new source and destination MAC addresses for the next hop. The MAC address the reader's router sees is the laptop's. The MAC address GitHub's server sees is its own upstream router's. This is the single most important fact about MAC addresses and the reason section 23.10 exists.

23.7.6 WORDS - remember these

1. MAC address — the hardware address of a network interface — a 48-bit EUI-48 identifier used for layer 2 delivery.
2. OUI — the manufacturer part of the address — organizationally unique identifier, the first 24 bits, allocated by the IEEE.
3. Unicast — addressed to one device — I/G bit clear in the first octet.
4. Multicast — addressed to a group — I/G bit set, with reserved prefixes 01:00:5E for IPv4 and 33:33 for IPv6.
5. Broadcast — addressed to everyone on the link — FF:FF:FF:FF:FF:FF, all bits set.

6. Locally administered — an address somebody made up — U/L bit set, not guaranteed unique outside the local link.
7. MAC randomization — your device using a fake address for privacy — rotating locally administered addresses, default on iOS 14+ and Android 10+.
8. EUI-64 — the 64-bit version — extended unique identifier used by IPv6 SLAAC and other technologies.

23.8 Hubs, switches and how a switch learns

23.8.1 PLAIN - in simple words

1. In the middle of a star-shaped network sits a box that everything plugs into.
2. There have been two kinds of box, and the difference between them is the biggest single improvement in local networking history.
3. A **hub** is the dumb one. Anything arriving on any port is copied out of every other port, immediately, with no thought.
4. So every device hears everything. The whole hub is one shared medium, and collisions happen exactly as on the old coax cable.
5. A **switch** is the smart one. It looks at where the frame is going and sends it out of one port only.
6. To do that it needs to know which device is on which port. Nobody tells it. It works this out by itself, by watching.
7. Every frame carries a source address. When a frame comes in on port 3 saying it is from address X, the switch writes down "X is on port 3".
8. It builds a table this way, purely by listening. This is called **learning**.
9. If a frame arrives for an address the switch has not learned yet, it does the only safe thing: it sends it out of every port except the one it came in on. That is **flooding**.
10. Hubs have been obsolete since roughly 2005. You will only meet one in a museum or a very old cupboard.

23.8.2 PLAIN - a picture in your head

1. Picture a hotel with a single letterbox for the whole building, and a porter.
2. The hub porter takes every letter and photocopies it into every room's pigeonhole. Everyone reads everything and throws away what is not theirs.
3. The switch porter starts on day one knowing nothing, with a blank notebook.
4. A letter comes out of room 12 with "from: Anita" on it. The porter writes "Anita, room 12" in the notebook.
5. Later a letter arrives addressed to Anita. The porter checks the notebook and walks it straight to room 12.
6. A letter arrives for Vikram, who is not in the notebook yet. The porter has no choice: a copy goes to every room. Somebody will be Vikram.
7. When Vikram replies, his letter has "from: Vikram" on it, and the porter writes down his room. From then on, letters to Vikram go straight there.
8. If a guest checks out and a new one moves in, the notebook goes stale. So each entry is rubbed out if it has not been seen for a while.

Where this comparison breaks: the porter walks at human speed and can think. A switch must make this decision in a few hundred nanoseconds, for millions of frames a second, simultaneously on every port. It does it with dedicated silicon and a content-addressable memory, not by looking things up in a list. Also, the porter would notice two guests claiming the same name. A switch does not object; it simply moves the entry, which is exactly how one class of attack works.

23.8.3 PLAIN - a worked example

1. Three devices plug into a four-port switch. All addresses shortened for readability.

```
Port 1: laptop     AA:AA
Port 2: phone      BB:BB
Port 3: TV         CC:CC
Port 4: router     DD:DD   (192.168.0.1)
```

2. Step 0. The switch has just been powered on. Its table is empty.

```
MAC table: (empty)
```

3. Step 1. The laptop sends a frame to the router: source AA:AA, destination DD:DD, arriving on port 1.
4. The switch learns the source: AA:AA is on port 1. It looks up DD:DD, finds nothing, and floods the frame out of ports 2, 3 and 4.

```
MAC table:
  AA:AA -> port 1
```

5. Step 2. The router replies: source DD:DD, destination AA:AA, arriving on port 4.
6. The switch learns DD:DD is on port 4. It looks up AA:AA, finds port 1, and sends the frame only to port 1. The phone and TV never see it.

```
MAC table:
  AA:AA -> port 1
  DD:DD -> port 4
```

7. Step 3. The phone sends to the router: source BB:BB, destination DD:DD, on port 2.
8. The switch learns BB:BB on port 2, looks up DD:DD, finds port 4, forwards only there. No flooding at all this time.

```
MAC table:
  AA:AA -> port 1
  BB:BB -> port 2
  DD:DD -> port 4
```

9. The TV on port 3 has still never transmitted, so it is still unknown. Traffic for it will be flooded until it speaks once.
10. After three frames the switch has learned three of four devices and stopped flooding almost entirely. This is why switched networks scale and hubs do not.

23.8.4 PLAIN - what is really happening inside

1. The table is called the **MAC address table**, or the forwarding database, or the CAM table depending on who is speaking.
2. Each entry holds an address, a port number, a VLAN identifier and a timestamp.
3. Entries expire. The default ageing time is 300 seconds on most vendors. Every time a frame is seen from that address, the timer resets.
4. Ageing is what lets you unplug a laptop from one port and plug it into another. Actually, the switch relearns instantly on the first frame from the new port, and ageing only cleans up devices that have gone away entirely.
5. There are two ways to forward a frame, and they trade latency against safety.
6. **Store-and-forward** reads the entire frame into memory, checks the FCS, and only then forwards it. Corrupt frames never leave the switch.
7. **Cut-through** starts forwarding as soon as it has read the destination address, six bytes in. Far lower latency, but it will happily forward a frame that turns out to be corrupt.

8. A **collision domain** is a set of ports that can interfere with each other. On a switch, each port is its own collision domain, and on full duplex that domain has no contention at all.
9. A **broadcast domain** is the set of ports a broadcast frame reaches. A whole switch is one broadcast domain by default, and so is a whole stack of switches wired together.
10. Splitting a broadcast domain requires either a router or a VLAN, which is section 23.9.

23.8.5 TECHNICAL - the engineer's version

1. The correct standards term for a switch is a **MAC bridge**, defined by IEEE 802.1D. "Switch" is universal industry usage, that is, a convention.
2. Learning is defined in 802.1D as the learning process, and flooding of unknown unicast is required behaviour, not an optimization.
3. Lookups use ternary content-addressable memory (TCAM) or hash-based exact-match tables. TCAM does a full parallel comparison in one clock cycle, at a substantial cost in silicon area and power.
4. Table sizes are a real limit. A small unmanaged switch may hold 1000 to 8000 entries. An access switch holds 16k to 32k. A data centre spine switch holds 256k or more.
5. When the table is full, new entries cannot be learned and all traffic to unknown addresses is flooded. **MAC flooding** is an attack that deliberately fills the table with bogus source addresses to turn a switch into a hub and enable eavesdropping. The defence is port security, limiting learned addresses per port.

Forwarding mode	Latency at 1 GbE	Corrupt frames
Store-and-forward	About 5 to 12 us	Dropped
Cut-through	About 1 to 3 us	Forwarded
Fragment-free	About 1 us plus 64B	Mostly dropped

6. Fragment-free is a compromise: wait for the first 64 bytes, which is where collision fragments would appear, then start forwarding.
7. Modern data centre switches quote port-to-port latency of 300 to 800 nanoseconds in cut-through mode. Store-and-forward latency scales with frame size, because the whole frame must arrive first; a 1518-byte frame at 1 Gbit/s takes about 12 us just to receive.
8. What a **managed** switch adds over an unmanaged one: VLANs (802.1Q), spanning tree (802.1D and successors), link aggregation (802.1AX, formerly 802.3ad), port mirroring for capture, quality of service queues (802.1p), 802.1X port authentication, LLDP neighbour discovery (802.1AB), SNMP monitoring, and per-port statistics and rate limiting.
9. Link aggregation is directly relevant to this book's running example. The traceroute in the reader's session showed interfaces named `ae66-0`, `be23` and `po22`. Those prefixes mean aggregated Ethernet, bundle Ethernet and port-channel: three vendor words for several physical links bonded into one logical link. Those are naming conventions, not standards.
10. That bonding is also why hops 5, 6, 8, 9 and 11 in the reader's traceroute showed more than one address for a single hop. Packets in the same trace took different member links of the same bundle. That is load balancing working correctly, not a fault.
11. Observation commands, vendor-dependent: `show mac address-table` on Cisco and Arista, `show ethernet-switching table` on Juniper, `bridge fdb show` on Linux, and `ifconfig bridge0` on macOS for the built-in bridge.

23.8.6 WORDS - remember these

1. Hub — repeats everything to everyone — a multiport repeater, one collision domain, obsolete since the early 2000s.
2. Switch — sends each frame to one port — a MAC bridge per IEEE 802.1D, learning and forwarding per destination address.

3. MAC address table — the switch's notebook — the filtering database mapping address and VLAN to egress port, with an ageing timer.
4. Learning — writing down who is where — populating the filtering database from observed source addresses.
5. Flooding — sending to everyone because you do not know — forwarding unknown unicast, broadcast and unregistered multicast to all ports in the VLAN.
6. Collision domain — where two talkers can clash — the set of ports sharing one contended medium; one per port on a switch.
7. Broadcast domain — how far a broadcast reaches — the set of ports a broadcast frame is flooded to, bounded by a router or a VLAN edge.
8. Store-and-forward — read it all, check it, then send — full-frame buffering with FCS validation before egress.
9. Cut-through — start sending after the address — forwarding begins after the destination MAC is read, propagating errored frames.
10. Managed switch — one you can configure — a bridge supporting 802.1Q, spanning tree, aggregation, mirroring and management protocols.

23.9 VLANs and spanning tree, briefly

23.9.1 PLAIN - in simple words

1. One switch is one network. Every device on it can reach every other device, and every broadcast reaches everyone.
2. Often that is not what you want. An office may want staff computers, guest WiFi, security cameras and phones kept apart from each other.
3. You could buy four separate switches and four sets of cables. That is expensive and inflexible.
4. Instead you tell one switch to pretend to be four switches. Each pretend switch is called a **VLAN**, a virtual local area network.
5. Ports are assigned to VLANs. A port in VLAN 10 cannot reach a port in VLAN 20, even though they are in the same physical box.
6. To get between VLANs you need a router. That is the point: it gives you a place to apply rules.
7. Now the second idea. A network with two paths between the same two switches has a **loop**.
8. Loops are wonderful for reliability and catastrophic without protection, because a broadcast frame goes round the loop forever.
9. There is no field in an Ethernet frame that counts hops, so nothing ever stops it. It multiplies at every switch and the network dies in seconds.
10. **Spanning tree** is the protocol that finds loops and switches off just enough links to break them, while keeping those links ready as backups.

23.9.2 PLAIN - a picture in your head

1. Picture an office floor with one big open room.
2. VLANs are movable partition walls. The floor is unchanged, but people in one section cannot walk into another without going through a door with a guard on it. The guard is the router.
3. To move a person from one section to another you do not rewire anything. You change which section their desk is assigned to. That is reassigning a port to a VLAN.
4. Now picture the announcement system. A message sent in section A is repeated by every loudspeaker in section A only.
5. Now the loop. Imagine two corridors joining the same two rooms, and a rule that any announcement heard must be repeated into every other corridor.
6. Someone makes one announcement. It goes round both corridors, arrives back, gets repeated, and now there are two. Then four. Then eight.

7. Within seconds nobody can hear anything else. That is a **broadcast storm**.
8. Spanning tree is a supervisor who walks the building once, works out the corridors, and locks exactly one door so no circular route exists.

Where this comparison breaks: a locked door stays locked. Spanning tree keeps watching, and if the open corridor collapses it unlocks the spare door within seconds. Also, the supervisor is not a person: every switch runs the algorithm independently and they reach the same answer by exchanging small messages, with no central authority. That distributed agreement is the clever part.

23.9.3 PLAIN - a worked example

1. A small office, one 24-port switch, three VLANs.

VLAN	Name	Ports
10	Staff	1 to 12
20	Guest WiFi	13 to 18
30	Cameras	19 to 23
all	Uplink to router	24

2. Ports 1 to 23 are **access ports**. Each belongs to exactly one VLAN. The device plugged in knows nothing about VLANs at all.
3. Port 24 is a **trunk port**. It carries all three VLANs to the router over one cable.
4. For that to work, frames leaving port 24 must say which VLAN they belong to. So a 4-byte tag is inserted into each frame.
5. A camera on port 19 sends a frame. It is untagged, a plain 64-byte frame.
6. The switch forwards it towards port 24 and inserts a tag saying VLAN 30. The frame is now 68 bytes.
7. The router receives it, sees VLAN 30, and applies the camera policy: allowed to reach the recorder, not allowed to reach the internet.
8. A guest laptop on port 15 broadcasts. That broadcast reaches ports 13 to 18 and the trunk, and never touches ports 1 to 12. The staff never see it.
9. Without VLANs, one broadcast domain of 23 ports. With VLANs, three broadcast domains sharing one box.

23.9.4 PLAIN - what is really happening inside

1. The tag is four bytes inserted immediately after the source MAC address.
2. The first two bytes are always `0x8100`. That is the EtherType that says "a VLAN tag follows".
3. The next two bytes hold three things: three bits of priority, one bit that is almost always zero, and twelve bits of VLAN identifier.
4. Twelve bits gives 4096 values, of which 0 and 4095 are reserved, so 4094 usable VLANs.
5. Because the tag is inserted, the switch must recompute the frame check sequence. Adding a tag changes the frame, so the old CRC is no longer valid.
6. An access port strips the tag on the way out, so end devices never see it.
7. Now spanning tree. Every switch sends small messages called BPDUs out of every port every two seconds.
8. Each BPDU carries the sender's identity and its cost to reach the switch it currently believes is the root of the tree.
9. Through this gossip, all switches agree on one **root bridge**: the switch with the numerically lowest bridge identifier.
10. Every other switch works out its own cheapest path to the root, keeps that port forwarding, and blocks any other port that would create a second path.

11. A blocked port still listens to BPDUs. If they stop arriving, the topology has changed, and the blocked port is brought back into service.

23.9.5 TECHNICAL - the engineer's version

1. VLAN tagging is IEEE 802.1Q, first published in 1998 and now folded into 802.1Q-2022. The tag structure:

Part	Bits	Meaning
TPID	16	Always 0x8100
PCP	3	Priority, 802.1p
DEI	1	Drop eligible indicator
VID	12	VLAN ID, 1 to 4094

2. A tagged frame's maximum size is 1522 octets instead of 1518. Older equipment that counted 1518 as an absolute maximum reported these as "baby giants".
3. The **native VLAN** on a trunk is carried untagged, by convention VLAN 1. This is the basis of VLAN hopping via double tagging, so best practice is to set the native VLAN to an unused ID and tag everything.
4. 802.1ad (2005), "QinQ" or provider bridging, stacks a second tag with TPID 0x88A8 so a carrier can transport a customer's whole VLAN space.
5. Spanning Tree Protocol was invented in 1985 at Digital Equipment Corporation by Radia Perlman, who described it in her paper "An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN" and in the poem "Algorhyme". IEEE published it as 802.1D in 1990.
6. Classic STP port states are blocking, listening, learning, forwarding and disabled. Convergence takes 30 to 50 seconds, because listening and learning each last one forward delay of 15 seconds by default.
7. Default timers: hello 2 seconds, forward delay 15 seconds, max age 20 seconds.
8. Rapid Spanning Tree Protocol, IEEE 802.1w (2001), cut the states to discarding, learning and forwarding, and converges within three hello times or within milliseconds of a detected physical link failure. It was merged into 802.1D-2004.
9. Multiple Spanning Tree Protocol, IEEE 802.1s (2002), runs one tree per group of VLANs so different VLANs can use different physical links. It was merged into 802.1Q-2005.
10. Shortest Path Bridging, IEEE 802.1aq (2012), and TRILL, RFC 6325 (2011), replace the tree with a link-state protocol so all links can carry traffic. In practice most large data centres skipped both and moved to layer 3 fabrics with ECMP, or to VXLAN overlays per RFC 7348 (2014).
11. Why a loop is catastrophic, precisely: the Ethernet header has no TTL or hop count field. An IP packet has a TTL that a router decrements, so an IP routing loop self-terminates after at most 255 hops. A layer 2 loop has no such stop condition. Broadcast and unknown-unicast frames replicate at every switch on every pass, so traffic grows geometrically until link utilization reaches 100 percent, typically within one to two seconds.
12. There is a second failure alongside the storm: **MAC table instability**. The same source address arrives on two ports alternately, so every switch rewrites that entry thousands of times a second, and legitimate unicast traffic gets flooded as well.
13. Protective features, all vendor implementation details rather than standards: BPDU guard (shut a port that receives a BPDU when it should not), root guard, loop guard, and storm control thresholds.
14. Home routers usually run no spanning tree at all. Plugging one cheap unmanaged switch into another twice, by accident, will take down a home or small office network completely, and this is a genuinely common fault.

23.9.6 WORDS - remember these

1. VLAN — one switch pretending to be several — a virtual LAN, an isolated broadcast domain identified by a 12-bit VID under IEEE 802.1Q.
2. Access port — a port for an ordinary device — an untagged port belonging to exactly one VLAN.
3. Trunk port — a port carrying many VLANs at once — a tagged port carrying 802.1Q-tagged frames for multiple VIDs.
4. Tag — the four bytes saying which VLAN — TPID 0x8100 plus PCP, DEI and VID, inserted after the source MAC.
5. Loop — two paths between the same switches — a cycle in the layer 2 topology with no hop count to stop frames.
6. Broadcast storm — the network drowning in copies of one frame — geometric replication of flooded frames around a loop.
7. Spanning tree — the protocol that breaks loops safely — IEEE 802.1D and its successors, electing a root bridge and blocking redundant ports.
8. BPDU — the small message switches gossip with — bridge protocol data unit, sent every hello interval to build and maintain the tree.
9. Root bridge — the switch everything measures distance from — the bridge with the lowest bridge ID, elected by all participants.

23.10 ARP: from an IP address to a MAC address

23.10.1 PLAIN - in simple words

1. Here is the problem this section solves, and it is the most important idea in the chapter.
2. Software works with IP addresses. The reader's laptop wants to send something to `192.168.0.1`, the router.
3. But the network card cannot send to an IP address. It can only send to a MAC address, the hardware address from section 23.7.
4. Nothing connects the two. An IP address does not contain a MAC address and a MAC address does not contain an IP address.
5. So the laptop has to ask. It shouts to the whole local network: "whoever has 192.168.0.1, tell me your hardware address".
6. The router hears its own IP address in the question and answers directly: "192.168.0.1 is at this MAC address".
7. That question-and-answer is **ARP**, the address resolution protocol.
8. The laptop writes the answer down in a small table so it does not have to ask again for every packet.
9. That table is the **ARP cache**, and entries in it expire after a few minutes.
10. ARP is only for the local network. To reach something far away, the laptop ARPs for the router and sends everything there. The router deals with the rest.

23.10.2 PLAIN - a picture in your head

1. You are in a large office and you have a note to hand to "the head of accounts". You do not know who that is or where they sit.
2. You stand up and call out to the whole floor: "who is the head of accounts?"
3. Everyone hears. Everyone except one person ignores you.
4. One person calls back: "that is me, third desk by the window."
5. You write that down on a sticky note on your monitor, and from then on you walk straight there.
6. A few weeks later the sticky note is out of date, because someone else took the job. So you throw the note away periodically and ask again.
7. Notice the danger. When you called out, anyone could have answered. Nobody checks credentials. The first confident reply wins.

Where this comparison breaks: in an office, a wrong answer would be noticed socially within a day. On a network, a false ARP reply is silently believed forever, or at least until the entry ages out, and the victim has no way at all to tell a true reply from a false one. ARP has no authentication of any kind. This is not a bug that was later fixed; it is the design, and the design is from 1982.

23.10.3 PLAIN – a worked example

1. This is the reader's own network. The laptop is at `192.168.0.107` and the router is at `192.168.0.1`. MAC addresses are illustrative, because the session record does not include them.

```
Step 1: the request, broadcast to everyone

Ethernet dest : FF:FF:FF:FF:FF:FF   (everyone)
Ethernet src  : 3C:22:FB:8A:41:D7   (the laptop)
EtherType     : 0x0806              (this is ARP)
  Opcode      : 1                   (request)
  Sender MAC  : 3C:22:FB:8A:41:D7
  Sender IP   : 192.168.0.107
  Target MAC  : 00:00:00:00:00:00   (unknown, that is
                                     the whole question)
  Target IP   : 192.168.0.1

Step 2: the reply, sent to one machine only

Ethernet dest : 3C:22:FB:8A:41:D7   (back to the laptop)
Ethernet src  : 5C:A6:E6:11:22:33   (the router)
EtherType     : 0x0806
  Opcode      : 2                   (reply)
  Sender MAC  : 5C:A6:E6:11:22:33
  Sender IP   : 192.168.0.1
  Target MAC  : 3C:22:FB:8A:41:D7
  Target IP   : 192.168.0.107
```

2. Notice the request is a broadcast and the reply is a unicast. The question must reach everyone. The answer only needs to reach the asker.
3. Notice also that the router learns the laptop's mapping for free, from the sender fields of the request. Both sides end up with a cache entry after one exchange.
4. Now the laptop can build a real frame. To send an IP packet to GitHub at `20.207.73.82`, it puts:
    1. Destination MAC: `5C:A6:E6:11:22:33`, the router.
    2. Source MAC: `3C:22:FB:8A:41:D7`, itself.
    3. Destination IP: `20.207.73.82`, GitHub.
    4. Source IP: `192.168.0.107`, itself.
5. Read that again, because it is the sentence that confuses everyone. The MAC addresses are local and change at every hop. The IP addresses are end to end and do not change.
6. Here is the reader's ARP cache, read on macOS:

```
$ arp -a
? (192.168.0.1) at 5c:a6:e6:11:22:33 on en0 ifscope [ethernet]
? (192.168.0.107) at 3c:22:fb:8a:41:d7 on en0 [ethernet]
? (192.168.0.255) at ff:ff:ff:ff:ff:ff on en0 ifscope [ethernet]
```

7. The question marks are hostnames macOS could not resolve. `en0` is the interface. `ifscope` means the entry is bound to that specific interface.

23.10.4 PLAIN - what is really happening inside

1. Before sending any IP packet, the machine asks one question: is this destination on my own local network, or somewhere else?
2. It answers that by comparing the destination IP against its own IP and subnet mask. The next chapter covers exactly how that comparison works.
3. If the destination is local, it ARPs for the destination itself.
4. If the destination is not local, it does not ARP for the destination at all. It ARPs for the **default gateway**, which here is `192.168.0.1`.
5. This is why every machine on the internet has a cache entry for its router and for nothing else outside its own street.
6. Packets waiting for an ARP reply sit in a small queue. If no reply comes, the request is retried a few times and then the packets are dropped and an error is reported to the application.
7. A **gratuitous ARP** is an unsolicited announcement: a machine broadcasts an ARP for its own address, which nobody asked for.
8. It has two honest uses. It tells everyone to update their cache when an address moves to new hardware, and it detects duplicate addresses at startup.
9. It also has a dishonest use. A machine can send gratuitous replies claiming to own the router's IP address, and every machine on the network will believe it.
10. That is **ARP spoofing**. The attacker now receives traffic meant for the router, can read it, change it, and pass it on. It is a textbook man-in-the-middle attack, and it requires only being on the same network.

23.10.5 TECHNICAL - the engineer's version

1. ARP is defined in RFC 826, published November 1982, by David C. Plummer. It is Internet Standard STD 37 and it has never been revised.
2. It rides directly on Ethernet with EtherType 0x0806. It is not carried in IP, so it is neither TCP nor UDP and has no port number.
3. The packet is 28 bytes for the IPv4-over-Ethernet case:

Field	Bytes	Value for IPv4
HTYPE hardware type	2	1 for Ethernet
PTYPE protocol type	2	0x0800 for IPv4
HLEN hardware length	1	6
PLEN protocol length	1	4
OPER operation	2	1 request, 2 reply
SHA sender hardware	6	Sender MAC
SPA sender protocol	4	Sender IP
THA target hardware	6	Target MAC
TPA target protocol	4	Target IP

4. 28 bytes is below the 46-byte minimum payload, so 18 bytes of padding are always added. Some old implementations leaked memory contents in that padding; this was named "Etherleak" and assigned CVE-2003-0001.
5. Cache timeouts are implementation details, not standards. On macOS and other BSD-derived systems the default is 1200 seconds, tunable with `sysctl net.link.ether.inet.arp_keep` on some versions. On Linux the relevant knobs are `net.ipv4.neigh.default.base_reachable_time_ms`, default 30000, and `gc_stale_time`, default 60 seconds. Cisco IOS defaults to 14400 seconds, four hours, which is far longer than most people expect.
6. Variants worth knowing by name: **proxy ARP** (RFC 1027, 1987), where a router answers on behalf of a host on another segment; **ARP probe** and **ARP announcement** (RFC 5227, 2008), used for duplicate

address detection; and **RARP** (RFC 903, 1984), the obsolete reverse lookup replaced by BOOTP and DHCP.

7. IPv6 does not use ARP at all. It uses Neighbor Discovery Protocol, RFC 4861 (2007), which runs over ICMPv6 and uses multicast rather than broadcast. The reader's session reported `IPv6: (none)`, so no NDP was in play in this case.
8. ARP spoofing defences: dynamic ARP inspection on managed switches, which validates ARP packets against the DHCP snooping binding table; static ARP entries for critical addresses; 802.1X port authentication; and simply encrypting everything so that intercepting the traffic gains the attacker little. TLS, covered later in Part F, is the real answer.
9. Commands to observe and manipulate ARP on macOS:

```
# show the whole table
arp -a

# show one entry, numeric only
arp -n 192.168.0.1

# delete one entry, forcing a fresh ARP
sudo arp -d 192.168.0.1

# delete every entry
sudo arp -a -d

# add a permanent static entry
sudo arp -S 192.168.0.1 5c:a6:e6:11:22:33

# watch the exchange live
sudo tcpdump -i en0 -e -n arp
```

10. On Linux the modern equivalents are `ip neigh show`, `ip neigh flush all` and `ip neigh replace`. The old `arp` command still works but is deprecated in favour of `ip`.
11. Diagnostic value: if `arp -a` shows no entry for your gateway, and deleting and re-pinging does not create one, the problem is beneath IP entirely. Your machine is not reaching the router at layer 2. That is a different fault from the reader's, whose ARP was clearly fine, since traffic reached hop 1 and twelve hops beyond it.
12. The honest version of "ARP is a local protocol": ARP messages are never forwarded by routers, so an ARP table only ever contains addresses on your own link. Any tool that appears to show you a remote machine's MAC address is showing you something else, usually a router's.

23.10.6 WORDS - remember these

1. ARP — asking who owns an IP address on this network — the Address Resolution Protocol, RFC 826, EtherType 0x0806.
2. ARP request — the broadcast question — opcode 1, sent to FF:FF:FF:FF:FF:FF with the target hardware address zeroed.
3. ARP reply — the direct answer — opcode 2, unicast back to the requester.
4. ARP cache — the notes you keep so you need not ask again — the neighbour table mapping IP to MAC, with a per-entry ageing timer.
5. Default gateway — the router you send everything distant to — the next hop for destinations outside your own subnet.
6. Gratuitous ARP — announcing yourself without being asked — an unsolicited ARP with sender and target protocol addresses equal.
7. ARP spoofing — lying about who owns an address — forging ARP replies to redirect traffic, enabling man-in-the-middle interception.

8. Proxy ARP — a router answering on someone else's behalf — RFC 1027 behaviour where a gateway replies for hosts it can reach.
9. Neighbor Discovery — the IPv6 replacement for ARP — RFC 4861, running over ICMPv6 with multicast solicitations.

23.11 DHCP: how the laptop got an address

23.11.1 PLAIN - in simple words

1. Nobody configured the reader's laptop with an IP address. It joined the WiFi and simply had one, in the `192.168.0.x` range.
2. Something handed it that address. That something is a **DHCP server**, and in a home it lives inside the router at `192.168.0.1`.
3. DHCP stands for dynamic host configuration protocol. It automates the whole job of joining a network.
4. There is a chicken-and-egg problem. To ask for an address, you must send a message. To send a message, you normally need an address.
5. DHCP solves it by letting the client send from the address `0.0.0.0`, meaning "no address yet", to the address `255.255.255.255`, meaning "everyone here".
6. The conversation takes four messages, and they have a memorable name: DORA.
7. **D**iscover: the client shouts "is there a DHCP server out there?"
8. **O**ffer: a server replies "yes, and you may have this address".
9. **R**equest: the client says "I accept that address" — and it says it out loud so any other servers know they were not chosen.
10. **A**cknowledge: the server confirms and records the assignment.
11. The address is not given away. It is lent, for a fixed period called a **lease**, and the client must renew before it runs out.

23.11.2 PLAIN - a picture in your head

1. Picture arriving at a large hotel that has no reception desk you can see.
2. You stand in the lobby and call out: "does anyone here give out rooms?" That is Discover.
3. A clerk calls back: "yes, room 107 is free, it comes with a floor plan, the number for the concierge, and it is yours for one day." That is Offer.
4. You call back, loudly enough for the whole lobby: "I will take room 107 from that clerk." That is Request. The loudness matters, because a second clerk who also had a room ready now knows to put it back.
5. The clerk writes your name in the register and hands you the key. That is Acknowledge.
6. Halfway through the day, you go and ask to extend. Usually you get it. If the clerk has gone, you keep trying, and when the time is nearly up you start asking any clerk at all.
7. If nobody at all answers your first call, you do not stand in the lobby forever. You pick a corner of the lobby to sit in and hope somebody notices. That corner is the `169.254.x.x` address, and it means you never got a room.

Where this comparison breaks: in a hotel, a second clerk handing out the same room would be caught by the register. On a network, two DHCP servers on the same segment genuinely can hand out conflicting addresses, and the usual cause is someone plugging a home router into an office network by mistake. There is no central register. There is only a convention that clients take the first offer and servers ping an address before offering it.

23.11.3 PLAIN - a worked example

1. This is the reader's laptop joining their home network. The address `192.168.0.107` is illustrative; the session record confirms only that it was in the `192.168.0.x` range.

```
1. DISCOVER   client -> broadcast
   Ethernet dst : FF:FF:FF:FF:FF:FF
```

```
   IP  src      : 0.0.0.0
   IP  dst      : 255.255.255.255
   UDP src port : 68
   UDP dst port : 67
   Contains     : client MAC, requested options

2. OFFER      server -> client
   IP  src      : 192.168.0.1
   IP  dst      : 255.255.255.255 (or the offered IP)
   UDP src port : 67
   UDP dst port : 68
   Contains     : yiaddr 192.168.0.107, mask, gateway,
                  DNS servers, lease time

3. REQUEST    client -> broadcast
   IP  src      : 0.0.0.0
   IP  dst      : 255.255.255.255
   UDP 68 -> 67
   Contains     : "I want 192.168.0.107 from 192.168.0.1"

4. ACK        server -> client
   IP  src      : 192.168.0.1
   UDP 67 -> 68
   Contains     : final confirmed values and lease time
```

2. Note the ports. The server always listens on 67. The client always listens on 68. They are fixed, and a client that used a random source port would never hear the reply.
3. Note that Request is broadcast even though the client now knows the server's address. That is deliberate, so other servers can withdraw their offers.
4. Suppose the lease is 86400 seconds, one day. Two timers are set from it.
5. T1, the renewal timer, fires at 50 percent: 43200 seconds, twelve hours in. The client asks its own server directly, by unicast, to extend.
6. T2, the rebinding timer, fires at 87.5 percent: 75600 seconds, twenty-one hours in. If the original server never answered, the client now broadcasts to any server at all.
7. If nothing answers by 86400 seconds, the client must stop using the address entirely and start again from Discover.
8. The reader can see all of this on macOS:

```
# current lease and everything the server sent
ipconfig getpacket en0

# just the router and DNS the server supplied
ipconfig getoption en0 router
ipconfig getoption en0 domain_name_server

# force a fresh lease
sudo ipconfig set en0 DHCP

# watch the whole DORA exchange live
sudo tcpdump -i en0 -n port 67 or port 68
```

23.11.4 PLAIN - what is really happening inside

1. A DHCP server does not just give an address. It gives a whole configuration, and this is the part people forget.
2. The standard things it hands out are the IP address, the subnet mask, the default gateway, the DNS servers, the domain name and the lease time.

3. It can also hand out time servers, TFTP boot servers, and dozens of other things, each identified by a number.
4. Here is a fact from the reader's session worth pausing on. Their DNS resolver was `1.1.1.1`, the Cloudflare public resolver, not the router.
5. That means either the DHCP server was configured to hand out `1.1.1.1`, or somebody overrode DHCP's suggestion on the laptop itself. Both are common. The session data does not say which.
6. Now the failure case, which is the diagnostically useful one.
7. If no DHCP server answers, the operating system assigns itself an address in the range `169.254.x.x`, picked at random and checked for clashes with ARP.
8. That is a **link-local** address. Two machines on the same cable with `169.254` addresses can talk to each other, and to nothing else at all.
9. So seeing `169.254.x.x` on your machine is not a random fault. It is a specific message, and the message is: "I never heard from a DHCP server".
10. That single observation narrows the fault enormously. The cable or radio link is probably up, since the machine is on a network. But nothing answered.
11. Common causes: the router's DHCP service is off or crashed, the pool is exhausted, the machine is on the wrong VLAN, or the DHCP relay is missing on a routed segment.
12. The reader never saw `169.254`. Their laptop held a valid `192.168.0.x` address throughout, and reached twelve hops out. So DHCP was not the fault here. This is what ruling something out looks like.

23.11.5 TECHNICAL - the engineer's version

1. Lineage: RARP, RFC 903 (June 1984). BOOTP, RFC 951 (September 1985). DHCP, RFC 1531 (October 1993), then RFC 1541, then the current RFC 2131 (March 1997). Options are in RFC 2132 (March 1997). DHCPv6 is RFC 3315 (2003), superseded by RFC 8415 (2018).
2. DHCP is backward compatible with BOOTP and reuses its packet format, which is why the fields still have BOOTP names such as `yiaddr` (your IP address), `siaddr` (next server), `giaddr` (relay agent address) and `chaddr` (client hardware address).
3. Message types are carried in option 53. The eight defined values are DISCOVER 1, OFFER 2, REQUEST 3, DECLINE 4, ACK 5, NAK 6, RELEASE 7 and INFORM
   8.
4. Options seen in nearly every real exchange:

Option	Name	Typical value
1	Subnet mask	255.255.255.0
3	Router	192.168.0.1
6	DNS servers	1.1.1.1
15	Domain name	localdomain
42	NTP servers	pool.ntp.org address
51	Lease time	86400 seconds
53	Message type	1 to 8
55	Parameter request	list of wanted options

5. Timers per RFC 2131: T1 defaults to 0.5 times the lease duration, T2 to 0.875 times it. Servers may override both with options 58 and 59. At T1 the client unicasts a REQUEST in RENEWING state. At T2 it broadcasts in REBINDING state.
6. A server may reply DHCPNAK if the requested address is no longer valid, for example after a laptop moves between networks and asks for its old address. The client must then restart from DISCOVER.
7. DHCP is UDP, not TCP. Ports 67 (server, "bootps") and 68 (client, "bootpc") are fixed by IANA. Both are below 1024, so a DHCP client needs privilege to bind port 68.

8. Broadcasts do not cross routers, so on any network larger than one segment a **DHCP relay agent**, also called an IP helper, forwards DISCOVER messages as unicast to a central server and records its own address in `giaddr` so the server knows which pool to use. RFC 3046 (2001) adds option 82, the relay agent information option, which carries the physical port the request came from.
9. Static assignment versus reservation, and the distinction matters operationally:
    1. A **static** address is configured on the client itself. The DHCP server does not know about it and may hand the same address to someone else.
    2. A **reservation**, or static lease, is configured on the server and keyed to the client's MAC address. The client still runs normal DORA and still gets the gateway, DNS and domain automatically.
    3. Reservations are almost always the right choice, because configuration stays in one place. Their weakness is MAC randomization, which changes the key the reservation is matched on.
10. IPv4 link-local is 169.254.0.0/16, defined in RFC 3927 (May 2005). The first and last 256 addresses, 169.254.0.0/24 and 169.254.255.0/24, are reserved. Microsoft calls the mechanism APIPA. The IPv6 equivalent is fe80::/10 per RFC 4291 (February 2006), and it is always present, not a fallback.
11. Security notes: DHCP has no authentication. A rogue server can hand out its own address as the gateway and intercept everything. The defence on managed switches is **DHCP snooping**, which permits server replies only on ports explicitly marked trusted, and builds a binding table that dynamic ARP inspection then uses.
12. Observation on Linux: `dhclient -v`, `networkctl status`, `journalctl -u systemd-networkd`, or the lease files under `/var/lib/dhcp/`. On macOS the lease lives under `/var/db/dhcpclient/leases/`.

23.11.6 WORDS - remember these

1. DHCP — the thing that gives your device an address automatically — Dynamic Host Configuration Protocol, RFC 2131, over UDP ports 67 and 68.
2. DORA — the four-step exchange — Discover, Offer, Request, Acknowledge.
3. Lease — the address is lent, not given — a time-limited allocation with renewal at T1 and rebinding at T2.
4. T1 and T2 — when to ask for more time — 50 percent and 87.5 percent of the lease duration by default.
5. Default gateway — the router address DHCP hands you — option 3, the next hop for non-local destinations.
6. Reservation — always giving the same device the same address — a server-side static lease keyed on the client MAC address.
7. Relay agent — how DHCP crosses a router — an IP helper that unicasts client broadcasts to a central server and fills in giaddr.
8. Link-local address — the address you give yourself when nothing answers — 169.254.0.0/16 per RFC 3927, reachable only on the same link.
9. DHCP snooping — the switch refusing to believe rogue servers — a security feature permitting server messages only on trusted ports.

23.12 The network interface card

23.12.1 PLAIN - in simple words

1. The **network interface card**, or NIC, is the piece of hardware that turns data in memory into signals on the medium, and back.
2. In a laptop it is not a card at all. It is a chip soldered to the board, and on WiFi it is a radio chip with an antenna.
3. It has four jobs. Frame the data, put the signal on the wire, filter incoming frames by address, and check the FCS.
4. It cannot do any of this alone. It needs a **driver**, which is the piece of operating system software that knows how to talk to that exact chip.

5. The card and the operating system share a set of memory slots called **ring buffers**. The name is because the slots are used in a circle, over and over.
6. There is one ring for outgoing frames and one for incoming.
7. To send, the operating system writes a frame into a free slot and tells the card. The card copies it out and marks the slot free again.
8. To receive, the card writes into a free slot and raises an **interrupt**, a hardware signal that tells the CPU to stop what it is doing and look.
9. At high speed, interrupting for every single frame would consume the whole CPU. So the card waits a little and interrupts once for a batch instead.
10. Normally the card throws away frames not addressed to it. If you turn that filtering off, it passes up everything it hears. That is how packet capture works.

23.12.2 PLAIN - a picture in your head

1. Picture a busy loading bay behind a shop, with a row of numbered bays around a circular driveway.
2. The staff inside load goods into a free bay and raise a small flag. The driver outside takes whatever bay has a flag, loads the lorry, and lowers the flag.
3. Deliveries arriving work the same way in reverse. The driver fills a free bay and rings a bell so the staff come out.
4. If lorries arrive every few seconds, ringing the bell each time means the staff never get anything else done.
5. So the driver waits until three lorries have arrived, or ten seconds have passed, then rings once. That is interrupt coalescing.
6. If all the bays are full because the staff are busy, arriving lorries are turned away. Those are dropped packets, and they are counted.

Where this comparison breaks: the flags in a real NIC are not checked by a person walking out to look. The card writes directly into main memory over the system bus, with no CPU involvement at all, using DMA. The CPU only finds out afterwards. Also, a lorry turned away can come back tomorrow. A dropped frame is gone, and it is entirely up to a higher-layer protocol to notice and resend it.

23.12.3 PLAIN - a worked example

1. Here is a real captured frame, decoded field by field. It is the first frame of an attempt to reach GitHub from the reader's laptop. MAC addresses are illustrative.

```
Raw bytes as tcpdump -xx would show them:

5c a6 e6 11 22 33 3c 22 fb 8a 41 d7 08 00 45 00
00 3c 1c 46 40 00 40 06 xx xx c0 a8 00 6b 14 cf
49 52 e3 f4 01 bb ...

Decoded:

5c a6 e6 11 22 33   Destination MAC, the router
3c 22 fb 8a 41 d7   Source MAC, the laptop
08 00               EtherType 0x0800, so IPv4
45                  IP version 4, header length 5 words
00                  Differentiated services, none set
00 3c               Total length 60 bytes
1c 46               Identification
40 00               Flags: Don't Fragment set
40                  TTL 64, the usual Linux/macOS start
06                  Protocol 6, which is TCP
xx xx               Header checksum
c0 a8 00 6b         Source IP 192.168.0.107
```

```
14 cf 49 52         Destination IP 20.207.73.82
e3 f4               Source port 58356
01 bb               Destination port 443
```

2. `14 cf 49 52` in decimal is 20, 207, 73, 82. That is exactly the address the reader's `curl` printed: `Trying 20.207.73.82:443...`.
3. `01 bb` in decimal is 443, the HTTPS port.
4. Notice the destination MAC is the router, not GitHub. GitHub is not on this network, so the frame is addressed locally and routed onwards.
5. Notice the TTL is 64. Every router that forwards this packet subtracts one. That mechanism is what makes traceroute possible, and it gets a chapter of its own later in Part F.
6. Everything from `45` onwards is the IP packet, which is the next chapter. Everything before it is this chapter.
7. In the reader's outage, frames exactly like this one went out and nothing ever came back. The frame was correct. The path was not.

23.12.4 PLAIN - what is really happening inside

1. A modern NIC does far more than move bytes. It does work that the CPU used to do, and this is called **offload**.
2. **Checksum offload** means the card computes the IP and TCP checksums in hardware as the frame goes out, and verifies them on the way in.
3. This has a confusing side effect for anyone capturing traffic. A capture taken on the sending machine shows outgoing checksums as invalid, because the card has not filled them in yet. That is normal, not a fault.
4. **Segmentation offload** means the operating system hands the card one huge block, say 64 kilobytes, and the card chops it into 1500-byte frames itself, generating all the headers.
5. That turns forty-odd trips through the network stack into one, which is a large saving at gigabit speeds and above.
6. The same trick in reverse merges many arriving small frames into one big one before handing it up. A capture on such a machine will show impossible 64-kilobyte "frames". They never existed on the wire.
7. **Promiscuous mode** turns off the address filter. The card passes up every frame it receives, regardless of destination.
8. On a switched network that gains you less than people expect, because the switch already only sends you frames for you, plus broadcasts and floods.
9. To capture everything on a switched network you need a **mirror port**, where the switch is configured to copy all traffic from certain ports to yours.
10. On WiFi, promiscuous mode is not enough either. You need **monitor mode**, which captures raw radio frames including management frames, and on many drivers it means disconnecting from the network to do it.

23.12.5 TECHNICAL - the engineer's version

1. The descriptor rings are arrays of small structures in host memory. Each descriptor holds a physical buffer address, a length and status bits. The driver owns some entries, the card owns others, and ownership is passed by writing a status bit and advancing a tail pointer.
2. Data movement is DMA, direct memory access, so the frame body never passes through the CPU. On systems with an IOMMU the addresses are translated and bounded, which is what stops a compromised NIC from reading arbitrary memory.
3. Interrupt coalescing is configured by two thresholds: a packet count and a time. Tuned with `ethtool -C eth0 rx-usecs 50 rx-frames 32` on Linux. Lower values reduce latency and raise CPU usage; higher values do the opposite.

4. Linux replaced pure interrupt-driven receive with **NAPI** in kernel 2.4.20 (2002). On the first interrupt the driver disables further receive interrupts and switches to polling, then re-enables them when the ring drains. This is what stops a machine from livelocking under a packet flood.
5. Offload features, with their Linux names:

Feature	Linux name	What it does
RX checksum	rx-checksumming	Verify in hardware
TX checksum	tx-checksumming	Compute in hardware
TCP segmentation	tso	Card splits large sends
Generic segmentation	gso	Kernel splits late
Large receive	lro	Card merges arrivals
Generic receive	gro	Kernel merges arrivals
Receive side scaling	rss	Spread flows over queues

6. Receive side scaling hashes each flow's addresses and ports to choose one of several receive queues, each bound to a different CPU core. Without it a single core would have to handle a whole 10 or 100 Gbit/s interface.
7. Modern high-speed NICs support SR-IOV, presenting many virtual functions to virtual machines directly, and kernel-bypass frameworks such as DPDK and AF_XDP that let user-space code poll the ring itself. That is how software routers reach line rate.
8. Packet capture on Unix goes through **BPF**, the Berkeley Packet Filter, described by Steven McCanne and Van Jacobson in the 1993 USENIX paper "The BSD Packet Filter: A New Architecture for User-level Packet Capture". The filter expression is compiled to bytecode and run in the kernel, so unwanted packets are discarded before they are copied to user space.
9. Linux later generalized this into eBPF, which now runs far more than packet filters, and into AF_PACKET with a memory-mapped ring for capture.
10. `tcpdump` was written at Lawrence Berkeley Laboratory starting in 1988 by Van Jacobson, Craig Leres and Steven McCanne. `libpcap`, the capture library underneath it, came from the same group and is what almost every capture tool uses today.
11. Wireshark began as Ethereal, written by Gerald Combs in 1998, and was renamed Wireshark in 2006 over a trademark issue. Its command-line form is `tshark`.
12. Useful capture commands:

```
# macOS: capture ARP and DHCP with Ethernet headers shown
sudo tcpdump -i en0 -e -n 'arp or port 67 or port 68'

# capture to a file for Wireshark, full frames
sudo tcpdump -i en0 -s 0 -w capture.pcap

# only traffic to the address curl was trying
sudo tcpdump -i en0 -n host 20.207.73.82

# show raw bytes including the Ethernet header
sudo tcpdump -i en0 -xx -c 1
```

13. On macOS, `tcpdump` on a WiFi interface will not show other stations' traffic even with `-p` disabled, because the radio decrypts only frames for you. Apple's `airport` utility historically enabled monitor mode, but it was removed in macOS 14.4 (2024); the supported route now is the Wireless Diagnostics application. That is an implementation detail and it has changed more than once.
14. Interface statistics worth reading before anything else: `netstat -i` on macOS, `ip -s link` on Linux, and `ethtool -S eth0` for the full vendor-specific counter set. Non-zero `Ierrs`, `Oerrs` or `rx_crc_errors` point at a physical problem: a bad cable, a dirty optic, or a duplex mismatch.

15. The honest version: a capture taken on the machine under test is not the truth about the wire. Offloads reshape it, the driver may drop frames before the filter, and the timestamps come from the host clock unless the NIC does hardware timestamping. For a definitive answer you capture from a mirror port or a passive tap on another machine.

23.12.6 WORDS - remember these

1. NIC — the hardware that connects you to the network — network interface controller, implementing the physical and MAC layers.
2. Driver — the software that knows this exact chip — the kernel module managing descriptor rings, interrupts and offload configuration.
3. Ring buffer — a circle of memory slots shared with the card — a descriptor ring with head and tail pointers and DMA buffer addresses.
4. DMA — the card writing to memory by itself — direct memory access, moving frame data without CPU involvement.
5. Interrupt coalescing — batching the "look at me" signals — delaying interrupts by packet count or time to trade latency for CPU cost.
6. Checksum offload — the card doing the arithmetic — hardware computation and verification of IP, TCP and UDP checksums.
7. Segmentation offload — handing the card one big block — TSO or GSO, splitting a large buffer into MTU-sized frames in hardware or late in the stack.
8. Promiscuous mode — accepting frames not addressed to you — disabling the MAC destination filter for capture.
9. Mirror port — a switch port that copies other ports' traffic — SPAN or port mirroring, required to capture on a switched network.
10. BPF — the filter that decides what to capture — Berkeley Packet Filter bytecode executed in the kernel, used by libpcap and tcpdump.

23.13 Putting the local network together

23.13.1 PLAIN - in simple words

1. We now have every piece. Let us watch the reader's laptop join the network from cold, in order, naming everything.
2. It starts with radio, because the reader is on WiFi. The card scans for networks, picks one, and authenticates and associates with the access point.
3. From that moment the laptop is on a network, but it has no address and knows nothing about it.
4. So it asks. It broadcasts a DHCP Discover from `0.0.0.0` to `255.255.255.255`.
5. The router answers with an Offer containing an address in the `192.168.0.x` range, the mask, the gateway `192.168.0.1`, and the DNS servers.
6. The laptop broadcasts a Request accepting it, and the router sends an Acknowledge. The laptop now has an address and a lease.
7. Before it uses that address, it usually checks nobody else has it, by sending an ARP probe for its own address and listening for a reply.
8. Now it wants to reach something. It compares the destination against its own subnet and finds the destination is elsewhere.
9. So it needs the router's hardware address. It broadcasts an ARP request for `192.168.0.1` and gets a reply.
10. Only now can the first real frame be built and sent. Everything up to this point was preparation, and all of it is beneath IP.

23.13.2 PLAIN - a picture in your head

1. Think of arriving in a new city where you know nobody.
2. First you have to get through the door of the building. That is joining the WiFi.
3. Then you need an address of your own, so anyone can reply to you. That is DHCP.
4. Then you check nobody else already lives at that address. That is the ARP probe.
5. Then you need to know which door leads out of the building. That is the default gateway.
6. Then you need to know the physical location of that door, not just its name. That is ARP for the gateway.
7. Only then can you post your first letter.

Where this comparison breaks: a person does all of this once and remembers it for years. A laptop redoes almost all of it every time it wakes from sleep, changes network, or the lease timer expires. It also does it again silently every few minutes as ARP entries age out. The steady state of a network is not stillness; it is this whole sequence running quietly, over and over.

23.13.3 PLAIN - a worked example

1. Here is the full sequence for the reader's laptop, from cold to first packet towards GitHub.

```
TIME  WHO        WHAT                          PROTOCOL
----  ---------  ----------------------------  ---------
t+0   laptop     scan for networks             802.11
t+1   laptop     authenticate, associate       802.11
t+2   both       4-way key handshake           802.11i
t+3   laptop     DHCP Discover, broadcast      DHCP/UDP68
t+3   router     DHCP Offer 192.168.0.107      DHCP/UDP67
t+3   laptop     DHCP Request, broadcast       DHCP/UDP68
t+3   router     DHCP Ack, lease granted       DHCP/UDP67
t+3   laptop     ARP probe for own address     ARP
t+4   laptop     ARP who has 192.168.0.1       ARP
t+4   router     ARP reply, here is my MAC     ARP
t+5   laptop     DNS query for github.com      DNS/UDP53
t+5   resolver   answer 20.207.73.82           DNS/UDP53
t+5   laptop     TCP SYN to 20.207.73.82:443   TCP
t+20  ...        nothing comes back at all     silence
```

2. Steps t+0 to t+4 are this chapter. Everything from t+5 belongs to later chapters in Part F, and each gets its own.
3. The DNS lookup went to `1.1.1.1`, the Cloudflare public resolver, and it worked. The name resolved to `20.207.73.82`, which is in a Microsoft-owned range; GitHub has been owned by Microsoft since 2018 and fronts traffic through Microsoft's network edge.
4. So name resolution succeeded and the local network worked perfectly.
5. The TCP connection attempt then received no response at all for fifteen seconds. No SYN-ACK, no reset, no ICMP unreachable.
6. What that proves: layers 1 and 2 were healthy end to end on the local segment, DHCP worked, ARP worked, and DNS worked.
7. What it merely suggests: the packets were being discarded silently somewhere beyond the local network. The evidence is consistent with a silent drop on that path. It is not consistent with the server being down, because the same site loaded instantly over mobile data on the same phone.
8. We will not name a culprit. The observation supports a path-specific silent drop and nothing more.

23.13.4 PLAIN - what is really happening inside

1. Here is the same journey drawn as a picture, with the layers each hop cares about.

```
   LAPTOP                ROUTER              THE INTERNET
   192.168.0.107         192.168.0.1
```

```
+----------+          +----------+
| app      |          |          |
| TCP      |          |          |
| IP       |--------->| IP       |------> hop 2 ...
| Ethernet |  frame   | Ethernet |
| radio    |<-------->| radio    |
+----------+  802.11  +----------+

The frame stops at the router. It is unwrapped,
the IP packet inside is read, and a brand new
frame is built for the next hop.

MAC addresses: change at every single hop.
IP addresses : stay the same all the way.
```

2. That last pair of lines is the single most important idea in local networking, and the reason MAC and IP both exist.
3. The reader's traceroute confirms it. Hop 1 was `192.168.0.1`, the router in the flat. Hop 2 was `172.31.0.17`, inside the ISP.
4. Between those two hops the frame was completely rebuilt, with different source and destination MAC addresses, on a different physical medium.
5. The IP packet inside was unchanged apart from the TTL being decremented by one and the header checksum being recomputed.
6. Hops 2 to 6 in the reader's trace were private addresses in the RFC 1918 ranges, belonging to the ISP's own core. `172.16.0.0/12` covers 172.16.x through 172.31.x, so `172.31.0.17`, `172.26.22.235` and `172.16.18.33` are all private.
7. Hop 3, `137.97.29.249`, was a public address belonging to the ISP.
8. From hop 7 the path entered Microsoft's backbone at `ntwk.msn.net`, went through Delhi, then Mumbai, then Pune, and then produced nothing.
9. All of that is layer 3 and above, and it is the subject of the next several chapters. This chapter ends where the frame ends: at the router in the flat.

23.13.5 TECHNICAL - the engineer's version

1. Sequence with the exact standards involved, for a WPA2 or WPA3 personal WiFi network:
    1. Probe request and response, or passive beacon reception. IEEE 802.11.
    2. Open system authentication, then association request and response. IEEE 802.11.
    3. 4-way handshake deriving the pairwise transient key. IEEE 802.11i, ratified 2004, marketed as WPA2. WPA3 (2018) replaces the pre-shared key exchange with SAE.
    4. DHCP DORA. RFC 2131, UDP 67 and 68.
    5. ARP announcement or probe for the assigned address. RFC 5227.
    6. ARP request for the default gateway. RFC 826.
    7. DNS query. RFC 1035, and later chapters cover it.
    8. TCP three-way handshake. RFC 9293, which in 2022 replaced RFC 793 from 1981.
2. On a wired port, replace the first three steps with autonegotiation, and optionally 802.1X authentication using EAP over LAN before the port is opened.
3. Everything above layer 2 in that list is deferred. IP addressing and subnets, NAT, DNS, routing and traceroute, TCP, and TLS each get a full chapter in Part F.
4. Diagnostic ordering follows the same sequence, and this is the practical payoff of the whole chapter. Test bottom-up and stop at the first failure.

Symptom	First thing to check
No link light	Cable, port, transceiver
Link up, no address	DHCP server, VLAN
169.254.x.x address	DHCP not answering
No ARP for gateway	Layer 2 path, VLAN
Gateway pings, nothing else	Routing or NAT
IP works, names do not	DNS
Name works, connect hangs	Path, filtering, MTU

5. The reader's fault sits on the last row. Every row above it had been passed.
6. A more precise statement of the observation: a TCP SYN was transmitted to `20.207.73.82` port 443 and no segment, no ICMP message, and no RST arrived within 15 seconds. Under RFC 9293 the client will retransmit the SYN with exponential backoff before giving up, which is why the wait was long and featureless rather than an immediate error.
7. An immediate "connection refused" would have been a TCP RST, proving something answered. An "unreachable" message would have been ICMP type 3, proving a router made a decision and told you. Silence proves only that nothing chose to speak.
8. Silence is the least informative failure mode in networking, and it is the most common one on the modern internet, because filtering devices are usually configured to drop rather than reject.

23.13.6 WORDS - remember these

1. Association — joining a WiFi network — the 802.11 exchange that binds a station to an access point before any data frames.
2. Default gateway — the way out of your network — the router address used for any destination outside your own subnet.
3. Encapsulation — wrapping data in a header at each layer — a TCP segment inside an IP packet inside an Ethernet frame.
4. Hop — one router-to-router step — a single layer 3 forwarding decision, with a complete new layer 2 frame each time.
5. TTL — the counter that stops packets looping forever — time to live, decremented at every hop, packet discarded at zero.
6. Silent drop — a packet discarded with no message — a firewall DROP rather than a REJECT, giving the sender no information.
7. Proven versus suggested — what an observation actually establishes — the discipline of separating direct evidence from inference, which is the whole method of Part F.

23.98 Common wrong ideas

1. Wrong: bandwidth and speed are the same thing. Right: bandwidth is capacity per second, latency is delay before anything arrives. Doubling capacity does not reduce delay by a single microsecond.
2. Wrong: a MAC address travels with the packet across the internet. Right: it is replaced at every single hop. The MAC addresses in a frame are only ever the two ends of one physical link.
3. Wrong: a switch and a hub do the same job. Right: a hub copies every frame to every port and shares one collision domain. A switch learns and forwards to one port, giving each port its own collision domain.
4. Wrong: collisions are a normal part of modern Ethernet. Right: on a switched full-duplex link collisions are impossible. If your interface counters show collisions, you have a duplex mismatch or a real hub.
5. Wrong: a 1500-byte MTU is a physical limit of the cable. Right: it is a number chosen in the 1980 DIX specification to limit how long one station could hold a shared coax cable, kept ever since for compatibility.

6. Wrong: category 6 cable makes your internet faster. Right: cable category sets the maximum a link can carry over 100 m. If your line delivers 100 Mbit/s, Cat 5e was already twenty times more than enough.
7. Wrong: ARP is part of IP. Right: ARP rides directly on Ethernet with EtherType 0x0806. It is not carried inside IP, has no port number, and never crosses a router.
8. Wrong: seeing `169.254.x.x` means the network card is broken. Right: it means no DHCP server answered. The link is probably up. That single observation rules out several faults and points at one.
9. Wrong: a traceroute ending in stars proves the destination is unreachable. Right: many routers rate-limit or block ICMP time-exceeded replies. Silence at the end of a trace is normal and proves nothing on its own.
10. Wrong: MAC address filtering secures a WiFi network. Right: addresses are broadcast in clear text in every frame header and can be changed in one command. Since MAC randomization became the phone default it does not even work as an inconvenience.
11. Wrong: fibre is faster because light is faster than electricity. Right: a signal moves through fibre at about two-thirds of the vacuum speed of light, similar to copper. Fibre wins on attenuation and bandwidth over distance, not on propagation speed.
12. Wrong: a packet capture on your own machine shows exactly what was on the wire. Right: segmentation and checksum offload reshape what you see. Frames larger than the MTU and invalid outbound checksums are normal artefacts.

23.99 Chapter summary in 20 lines

1. A network is machines sharing a medium, and every network must solve addressing, framing, error detection and medium sharing.
2. Topologies are bus, star, ring and mesh; modern local networks are stars and the internet core is a mesh.
3. A wire carries voltages, not numbers, and plain high-low signalling fails on clock recovery and DC balance.
4. Line codes fix this: Manchester doubles the bandwidth, 4B/5B and 8b/10b cost 25 percent, 64b/66b costs 3.1 percent, PAM-4 trades margin for symbols.
5. Twisting cancels interference because both wires spend equal time nearest the noise, and the receiver reads only the difference.
6. Cable categories are frequency ratings: Cat 5e at 100 MHz, Cat 6A at 500 MHz, Cat 8 at 2000 MHz, with 100 m channels except Cat 8's 30 m.
7. Fibre guides light by total internal reflection; single-mode goes tens of kilometres, multi-mode a few hundred metres, and SFP modules set the reach.
8. Bandwidth, throughput, latency and jitter are four different things, and the bandwidth-delay product explains why a fast link can still feel slow.
9. The reader's line was not slow. It was silent, which is a different fault entirely and needs a different diagnosis.
10. Ethernet began at Xerox PARC on 22 May 1973 with Metcalfe and Boggs at 2.94 Mbit/s, was standardized as IEEE 802.3 in 1985, and now runs to 800 Gbit/s.
11. CSMA/CD and the 64-byte minimum frame exist because of collision timing on a shared cable, and neither matters on a modern full-duplex link.
12. The frame is preamble 7, SFD 1, destination 6, source 6, EtherType 2, payload 46 to 1500, FCS 4, for 64 to 1518 bytes.
13. The 1500-byte MTU came from the 1980 DIX specification and has never changed; fragmentation costs headers, memory and all-or-nothing loss.
14. A MAC address is 48 bits: a 24-bit vendor OUI plus a device part, with bits for multicast and for locally administered addresses.
15. Phones and laptops now randomize their MAC addresses by default, which breaks MAC filtering and MAC-keyed DHCP reservations.

16. A switch learns which address is on which port by reading source addresses, floods when it does not know, and ages entries after about 300 seconds.
17. VLANs split one switch into several broadcast domains using a 4-byte 802.1Q tag, and spanning tree stops loops that would otherwise storm in seconds.
18. ARP maps an IP address to a MAC address by broadcast question and unicast answer, has no authentication at all, and never crosses a router.
19. DHCP hands out address, mask, gateway, DNS and more in four messages over UDP ports 67 and 68, with renewal at 50 percent and rebinding at 87.5 percent.
20. Everything in this chapter happens before a single IP packet leaves the building, and the next six chapters take that packet the rest of the way.

24

CHAPTER

KB

IP Addresses, Private and Public, NAT and Carrier-Grade NAT

24.0 What this chapter gives you

1. You will be able to say exactly what an IP address labels, and why it is not a label for a device and not a label for a person.
2. You will be able to convert any IPv4 address to binary by hand, and back again, without a calculator.
3. You will be able to read a CIDR prefix such as `/24` or `/12` and work out the network address, the broadcast address, the first and last usable host, and the exact host count.
4. You will be able to look at any address and say instantly whether it is public, private, loopback, link-local, multicast, or something special, and name the RFC that says so.
5. You will be able to explain why every home near you can use 192.168.0.1 at the same time without anything colliding.
6. You will be able to describe NAT precisely: what the translation table holds, what source NAT and destination NAT are, and what your home router really does, which is PAT.
7. You will be able to explain what a port is, why NAT would be impossible without ports, and what the real limit on simultaneous connections is.
8. You will be able to say what NAT breaks, why peer-to-peer software needs STUN, TURN and ICE, and why NAT is honestly a workaround rather than a design.
9. You will be able to read the reader's own traceroute, hop by hop, and say what the private addresses at hops 2 to 6 do and do not prove about carrier-grade NAT.
10. You will be able to run the exact macOS commands that reveal your address, your gateway, your mask and your public address, and decide in under a minute whether you are behind carrier-grade NAT.

24.1 What an IP address is

24.1.1 PLAIN – in simple words

1. An IP address is a label. It says where to deliver something on a network.
2. The thing it labels is a **network interface**: one connection point, one way in and out of a machine.
3. It does not label the machine. A machine can have many connection points.
4. Your laptop has a Wi-Fi radio. That is one interface. It has an Ethernet socket. That is another. Each gets its own address.
5. If you turn on a VPN, the operating system makes a pretend interface in software. That gets an address too.
6. The reader's own macOS machine showed several interfaces named `utun`. Those are software tunnels. Each one can carry its own address.
7. So the question "what is my IP address" has no single answer. The honest answer is "which interface, and seen from where".

8. An IP address also does not label a person. It labels a doorway. Many people can walk through the same doorway.
9. Addresses are borrowed, not owned. Most home machines are given an address for a while and handed a different one later.

24.1.2 PLAIN - a picture in your head

1. Think of a large office building with several entrances.
2. The building has a main street entrance, a loading bay at the back, and a staff door at the side.
3. Each entrance has its own number painted above it. Deliveries come to the number that matches the entrance.
4. The number is not the name of the company inside. The company could move out tomorrow and a different one moves in, keeping the same painted number.
5. The number is not the name of any person inside either. Hundreds of people use the same doorway every day.
6. If the building opens a new temporary door for a festival, that door gets its own number for the week. That is what a VPN interface is like.

Where this comparison breaks: painted door numbers are stable for decades and tied to a physical place you can walk to. IP addresses are handed out for hours or days, can be moved between cities in minutes, and one address can be served by machines on three continents at once. A door number also cannot be shared by a thousand separate buildings at the same time, which is exactly what private addresses do.

24.1.3 PLAIN - a worked example

1. An IPv4 address is 32 switches in a row. Each switch is on or off, 1 or 0.
2. Writing 32 ones and zeros is horrible for humans, so we chop them into four groups of 8 and write each group as an ordinary number.
3. A group of 8 bits can hold 0 to 255. That is why every part of an address stays between 0 and 255.
4. The four numbers are written with dots between them. That is called **dotted quad** notation.
5. Let us convert the reader's own router address, 192.168.0.1, by hand.
6. Each of the 8 bits has a value. From left to right they are 128, 64, 32, 16, 8, 4, 2, 1. You take the biggest value that fits, subtract, and repeat.

Part	Working	Binary
192	128 + 64	11000000
168	128 + 32 + 8	10101000
0	nothing	00000000
1	1	00000001

7. So 192.168.0.1 in full is:

```
11000000 10101000 00000000 00000001
```

8. Now the address the reader's machine got for github.com, 20.207.73.82.

Part	Working	Binary
20	16 + 4	00010100
207	128+64+8+4+2+1	11001111
73	64 + 8 + 1	01001001
82	64 + 16 + 2	01010010

9. So 20.207.73.82 in full is:

```
00010100 11001111 01001001 01010010
```

10. Going backwards is just addition. Take 11001111, add up the columns that hold a 1: 128 + 64 + 8 + 4 + 2 + 1 = 207.
11. There are 32 bits, so there are 2 multiplied by itself 32 times possible addresses. That is 4,294,967,296, usually said as about 4.29 billion.

24.1.4 PLAIN - what is really happening inside

1. When your machine sends data, it wraps it in a small header, like an address label stuck on a parcel.
2. Inside that header are two fixed slots: 32 bits for where it came from, and 32 bits for where it is going.
3. The dots are not in the header. The dots exist only on paper and on screens for humans. On the wire it is 32 raw bits and nothing else.
4. Every router that touches the parcel reads the destination slot, looks it up in a table, and decides which cable to push it down next.
5. The router does not open the parcel. It does not care what is inside. It reads the label and forwards.
6. The source slot matters for the reply. Without it the far end would have no idea where to send anything back.
7. This is why a wrong source address is a real problem. If a machine lies about where it came from, replies go to the wrong place. That is called spoofing.
8. Your machine keeps a small list of its own interfaces and the address on each. When you open a connection, it picks the interface that can reach the destination and stamps that interface's address as the source.

24.1.5 TECHNICAL - the engineer's version

1. IPv4 is specified in RFC 791, published September 1981, by Jon Postel at the Information Sciences Institute. It is Internet Standard STD 5.
2. The address is a 32-bit unsigned integer carried in network byte order, which means big-endian, most significant byte first.
3. In a standard 20-byte IPv4 header, the source address occupies bytes 12 to 15 and the destination address occupies bytes 16 to 19, counting from zero.
4. Dotted quad notation is a presentation convention, not part of the protocol. RFC 791 itself writes addresses that way, but the wire format has no separators.
5. The same address has several equally valid written forms.

Form	192.168.0.1	20.207.73.82
Dotted quad	192.168.0.1	20.207.73.82
Decimal integer	3232235521	349129042
Hexadecimal	c0a80001	14cf4952
Old class	C	A

6. The total space is 2 to the power 32, exactly 4,294,967,296 addresses. Not all of them are usable, as section 24.4 shows.
7. An interface may hold more than one address. On Linux this is normal and unlimited in practice. On macOS you add extra addresses with `ifconfig en0 alias`.
8. Loopback is an interface too, normally `lo0` on macOS and `lo` on Linux, carrying 127.0.0.1.
9. Tunnel interfaces created by VPN software on macOS appear as `utun0`, `utun1` and so on. The reader's session showed several. Each can carry its own address and its own routes, which is exactly why a VPN can change which address the world sees without changing your Wi-Fi address at all.

10. Commands that observe this: `ifconfig -a`, `ipconfig getifaddr en0`, `networksetup -listallhardwareports` on macOS; `ip -4 addr show` on Linux.

```
ipconfig getifaddr en0        # just the IPv4 address of Wi-Fi
ifconfig en0 | grep 'inet '   # address, netmask, broadcast
ifconfig -a | grep utun       # the VPN tunnel interfaces
```

24.1.6 WORDS - remember these

1. IP address — a delivery label for one connection point — a 32-bit unsigned integer in the IPv4 header, bytes 12 to 15 and 16 to 19.
2. Interface — one way in or out of a machine — a network endpoint, physical or virtual, that can hold addresses and routes.
3. Dotted quad — writing an address as four numbers with dots — the presentation format for a 32-bit IPv4 address, one decimal octet per byte.
4. Octet — one of the four numbers, 0 to 255 — an 8-bit field; the term is used instead of "byte" because early machines had bytes of other sizes.
5. Big-endian — biggest part written first — network byte order, the byte ordering all IP header fields use.
6. Spoofing — lying about where a packet came from — forging the source address field, mitigated by ingress filtering as described in BCP 38.

24.2 Networks and hosts: the split, the mask and CIDR

24.2.1 PLAIN - in simple words

1. An IP address is really two facts glued together.
2. The left part says which network you are on. The right part says which machine you are within that network.
3. The glue point is not fixed. It can sit anywhere along the 32 bits.
4. Something has to say where the split is. That something is the **subnet mask**.
5. A mask is another 32-bit pattern. Every 1 in it means "this bit belongs to the network part". Every 0 means "this bit belongs to the machine part".
6. The ones always come first and never have gaps. So a mask is fully described by counting them.
7. Counting them is the whole idea of CIDR notation. You write a slash and the number of network bits. `/24` means the first 24 bits are network.
8. So 192.168.0.1/24 means: the network is the first 24 bits, and 8 bits are left over to number machines.
9. Two addresses are on the same network when their network parts match. That single fact decides whether your machine talks to something directly or hands it to the router.

24.2.2 PLAIN - a picture in your head

1. Think of a phone number written as an area code plus a local number.
2. The area code says which town. The local number says which phone in that town.
3. If you are calling someone in your own town, you dial the local part only. You do not need the area code.
4. If they are in a different town, you dial the whole thing, and the call goes through the long-distance exchange first.
5. Your machine does the same test. Same network part means "shout directly down the wire". Different network part means "give it to the router".
6. The router is the long-distance exchange. It is the only way out.

Where this comparison breaks: phone area codes are a fixed number of digits and map to real geography. A CIDR prefix can be any number of bits from 0 to 32, and it maps to whoever was allocated it, which can be a company with offices in ten countries. Phone numbers also have no equivalent of the broadcast address, and no equivalent of a mask you can change on your own equipment.

24.2.3 PLAIN - a worked example

1. Take the reader's own home network, written as 192.168.0.0/24.
2. /24 means the first 24 bits are the network. In binary:

```
address 192.168.0.1  11000000 10101000 00000000 00000001
mask    /24          11111111 11111111 11111111 00000000
                     |------ network -----------|- host -|
```

3. The **network address** is what you get when every host bit is 0. Here the last 8 bits go to zero, giving 192.168.0.0.
4. The **broadcast address** is what you get when every host bit is 1. The last 8 bits become 11111111, which is 255, giving 192.168.0.255.
5. The first usable host is the network address plus one: 192.168.0.1. That is the reader's router.
6. The last usable host is the broadcast address minus one: 192.168.0.254.
7. The total addresses are 2 to the power 8, which is 256. Two are reserved for network and broadcast, so 254 machines can be numbered.
8. Now take 137.97.0.0/16, the classical block containing the reader's hop 3 at 137.97.29.249.
9. /16 means the first 16 bits are the network, so the whole last two octets are host bits.

Item	192.168.0.0/24	137.97.0.0/16
Mask	255.255.255.0	255.255.0.0
Network address	192.168.0.0	137.97.0.0
Broadcast	192.168.0.255	137.97.255.255
First usable	192.168.0.1	137.97.0.1
Last usable	192.168.0.254	137.97.255.254
Total addresses	256	65,536
Usable hosts	254	65,534

10. Check that 137.97.29.249 really falls inside 137.97.0.0/16. Compare the first 16 bits: 137.97 matches 137.97. It does.
11. Honest note: 137.97.0.0/16 is the historical block boundary. We are using it as a clean worked example. The actual prefix the reader's ISP announces to the internet could be smaller, such as a /19 or /22, carved out of it. The arithmetic is the same either way.

24.2.4 PLAIN - what is really happening inside

1. When your machine wants to send to some address, it does one calculation first, and it does it for every single packet.
2. It takes the destination address and lines it up with its own mask.
3. It performs a bitwise AND. That means: for each bit position, the answer is 1 only if both are 1.
4. ANDing an address with its mask wipes out the host bits and leaves the network number.
5. It does the same to its own address, and compares the two results.
6. If they match, the destination is a neighbour. The machine finds the neighbour's hardware address and sends the frame straight to it.
7. If they do not match, the destination is elsewhere. The machine sends the frame to the router instead, using the router's hardware address but keeping the far-away destination IP inside.
8. Worked through: the reader's laptop wants 20.207.73.82.
9. 20.207.73.82 AND 255.255.255.0 gives 20.207.73.0.
10. The laptop's own 192.168.0.x AND 255.255.255.0 gives 192.168.0.0.
11. They differ, so the packet goes to 192.168.0.1, the router. This is the exact reason hop 1 of the reader's trace is the router.
12. The router then repeats the same test against its own, much larger, table.

24.2.5 TECHNICAL - the engineer's version

1. CIDR, Classless Inter-Domain Routing, is defined in RFC 4632, August 2006, which is BCP 122. It obsoletes RFC 1519 and RFC 1518, both published in September 1993.
2. A prefix is written `address/length`. The length is the count of leading 1 bits in the mask. Masks must be contiguous; discontiguous masks were legal in some ancient equipment and are not permitted now.
3. The number of addresses in a prefix of length n is 2 to the power (32 - n).
4. Usable host count is that minus 2 for any prefix shorter than /31, because the all-zeros and all-ones host patterns are the network identifier and the directed broadcast.
5. RFC 3021, December 2000, permits /31 on point-to-point links, where both addresses are usable and there is no broadcast. Service provider backbones use /31 heavily to save address space.
6. A /32 is a single host route, used for loopbacks on routers and for advertising one address.
7. Full reference table, /8 to /32:

Prefix	Mask	Addresses	Usable hosts
/8	255.0.0.0	16,777,216	16,777,214
/9	255.128.0.0	8,388,608	8,388,606
/10	255.192.0.0	4,194,304	4,194,302
/11	255.224.0.0	2,097,152	2,097,150
/12	255.240.0.0	1,048,576	1,048,574
/13	255.248.0.0	524,288	524,286
/14	255.252.0.0	262,144	262,142
/15	255.254.0.0	131,072	131,070
/16	255.255.0.0	65,536	65,534
/17	255.255.128.0	32,768	32,766
/18	255.255.192.0	16,384	16,382
/19	255.255.224.0	8,192	8,190
/20	255.255.240.0	4,096	4,094
/21	255.255.248.0	2,048	2,046
/22	255.255.252.0	1,024	1,022
/23	255.255.254.0	512	510
/24	255.255.255.0	256	254
/25	255.255.255.128	128	126
/26	255.255.255.192	64	62
/27	255.255.255.224	32	30
/28	255.255.255.240	16	14
/29	255.255.255.248	8	6
/30	255.255.255.252	4	2
/31	255.255.255.254	2	2 point-to-point
/32	255.255.255.255	1	1 host route

8. Routers select a next hop by **longest prefix match**: among all routes that contain the destination, the one with the most network bits wins. A /24 beats a /16 which beats the default route 0.0.0.0/0.
9. VLSM, Variable Length Subnet Masking, means different prefix lengths inside one organization. It became possible once routing protocols carried the mask with the prefix, which OSPF and BGP-4 do and RIP version 1 did not.
10. Aggregation is the point of CIDR: an ISP holding 137.97.0.0/16 announces one route to the world instead of 256 separate /24 routes. Without this the global routing table would have collapsed under its own size in the 1990s.

11. Observation commands: `ipcalc 192.168.0.0/24`, `sipcalc`, `netstat -rn` on macOS to see the routing table, `route -n get 20.207.73.82` to ask macOS which route and gateway it would pick for one destination.

```
route -n get 20.207.73.82
#   route to: 20.207.73.82
# destination: default
#     gateway: 192.168.0.1
#   interface: en0
```

24.2.6 WORDS - remember these

1. Subnet mask — the pattern saying which part is the network — a 32-bit value of contiguous 1 bits followed by 0 bits, ANDed with an address.
2. CIDR notation — the slash number — prefix length in bits, defined in RFC 4632, replacing the class system.
3. Network address — the address with all host bits 0 — the prefix identifier, not assignable to a host.
4. Broadcast address — the address with all host bits 1 — the directed broadcast for that subnet, per RFC 919.
5. Bitwise AND — keep only bits that are 1 in both — the masking operation used for every local-versus-remote forwarding decision.
6. Longest prefix match — the most specific route wins — the forwarding rule that picks the entry with the greatest prefix length.
7. VLSM — different sized subnets in one network — variable length subnet masking, requires classless routing protocols.
8. Aggregation — announcing one big block instead of many small ones — route summarization, the mechanism that keeps the global BGP table finite.

24.3 Public and private addresses

24.3.1 PLAIN - in simple words

1. Some addresses are meant for the open internet. Anyone anywhere can reach them. Those are **public** addresses.
2. Some addresses are set aside for use inside one building or one home. Those are **private** addresses.
3. A private address only means something inside its own network. Outside, it means nothing.
4. Three blocks of addresses were reserved for private use in 1996, in a document called RFC 1918.
5. The three blocks are: everything starting 10, a range starting 172.16 up to 172.31, and everything starting 192.168.
6. The reader's router is 192.168.0.1. That is in the third block. So is almost every home router in the world.
7. Your neighbour's router is very likely also 192.168.0.1 or 192.168.1.1. That causes no problem at all.
8. It causes no problem because private addresses never travel on the public internet. Routers out there refuse to carry them.
9. So the two 192.168.0.1 addresses never meet. They live in separate sealed worlds.
10. Something must translate between the sealed world and the open one. That something is NAT, and it is section 24.6.

24.3.2 PLAIN - a picture in your head

1. Think of the extension numbers in an office phone system.
2. Inside the office you dial 101 and reach reception. Extension 101 exists in that office and nowhere else.
3. Every office in the city has an extension 101. There is no clash, because you cannot dial an extension from outside.
4. From outside you dial the single main number of the building. Only that number is listed publicly.

5. The main number is the public address. The extensions are the private addresses.
6. The switchboard is the router doing NAT. It is the only thing that knows both worlds.

Where this comparison breaks: a human switchboard operator can put an incoming call through to any extension on request. A home router cannot do that by default. Nobody outside can ask for an extension, because the router has no idea which internal machine an unexpected incoming packet is meant for. That single difference is the source of most of the pain in section 24.8.

24.3.3 PLAIN - a worked example

1. Here are the three private blocks with their exact boundaries.

Block	Range	Addresses
10.0.0.0/8	10.0.0.0 to 10.255.255.255	16,777,216
172.16.0.0/12	172.16.0.0 to 172.31.255.255	1,048,576
192.168.0.0/16	192.168.0.0 to 192.168.255.255	65,536

2. The middle one confuses people, so let us prove its range.
3. 172.16.0.0/12 fixes the first 12 bits. The first octet, 172, uses 8 of them. That leaves 4 more, taken from the second octet.
4. In binary, 172 is 10101100. The second octet 16 is 00010000. The first four bits of the second octet are 0001, and those are frozen.
5. The remaining four bits of the second octet are free. They range from 0000 to 1111. So the second octet runs from 00010000 to 00011111, which is 16 to 31.
6. Therefore the block covers 172.16.x.x all the way to 172.31.x.x. It does not cover 172.32.x.x, and it does not cover 172.15.x.x.
7. Now apply that to the reader's own traceroute. Hops 2 to 6 were:

```
2  172.31.0.17
3  137.97.29.249
4  172.26.22.235
5  172.16.18.33 / 172.26.14.75 / 172.26.22.234
6  172.26.14.75 / 172.16.18.33
```

8. 172.31.0.17 has second octet 31. That is inside 16 to 31. Private.
9. 172.26.22.235, 172.26.14.75 and 172.26.22.234 all have second octet 26. Inside 16 to 31. Private.
10. 172.16.18.33 has second octet 16, the very bottom edge of the block. Private.
11. 137.97.29.249 is not in any private block. 137 is not 10, not 172, not 192. Public.
12. So of the reader's first six hops, five are private and one is public. That single fact is the whole story of section 24.9.

24.3.4 PLAIN - what is really happening inside

1. There is nothing physically different about a private address. The bits look the same as any other bits.
2. The difference is entirely an agreement. Everyone agreed not to give these blocks to anybody as public addresses, and agreed that routers on the public internet would not carry them.
3. Two mechanisms enforce it. First, the address registries never allocate these blocks to anyone. Second, network operators filter them.
4. Filtering means a router is configured with a rule: if a route for 192.168.0.0/16 arrives from a neighbour, discard it. If a packet with a private source arrives from outside, drop it.
5. Those unwanted routes and packets are called **bogons**, short for bogus.
6. So a packet with source 192.168.0.24 sent onto the open internet would either be dropped, or would arrive somewhere and generate a reply that could never find its way home.

7. That is why the translation in NAT is not optional. A private address must be swapped for a public one before it leaves.
8. Inside a network, private addresses behave exactly like public ones. Routing, masks, broadcasts, all identical. Only the outside world treats them differently.
9. Large organizations use 10.0.0.0/8 precisely because it is huge: 16.7 million addresses, enough for a national telecom network's internal plumbing.

24.3.5 TECHNICAL - the engineer's version

1. RFC 1918, "Address Allocation for Private Internets", February 1996, is BCP 5. Authors include Yakov Rekhter and Robert Moskowitz. It updated the earlier RFC 1597 from March 1994.
2. The three blocks and their historical classful description:

Prefix	RFC 1918 name	Old class equivalent
10.0.0.0/8	24-bit block	one class A network
172.16.0.0/12	20-bit block	16 class B networks
192.168.0.0/16	16-bit block	256 class C networks

3. RFC 1918 hosts are described in the document as being in one of three categories, from category 1 needing no external access at all up to category 3 needing full external access through translation.
4. The complete authoritative list of special IPv4 blocks lives in the IANA IPv4 Special-Purpose Address Registry, whose framework is RFC 6890, April 2013, later corrected by RFC 8190, June 2017.
5. Filtering practice: operators drop RFC 1918 prefixes at the edge. This is a strong convention and appears in BCP 84 and in the Team Cymru bogon lists, but it is not enforced by the protocol itself. A misconfigured network can and does leak private prefixes.
6. Private addresses appearing inside a traceroute are entirely normal. Many carriers number their internal point-to-point links from 10.0.0.0/8 or 172.16.0.0/12 to conserve public space. The reader's hops 2, 4, 5 and 6 are exactly this.
7. Important precision: an RFC 1918 address seen in a traceroute is the address of the router interface that generated the ICMP time-exceeded reply. It is not the address your packet was carrying. Do not confuse the two. Section 24.9 depends on this distinction.
8. A private address can still be globally reachable through a tunnel, a VPN or an MPLS VPN, because those carry it inside another packet. The rule is about the public routing table, not about physics.
9. Overlap is a real operational problem. When two companies merge and both use 10.0.0.0/8, engineers have to renumber one side or deploy NAT between them, sometimes called twice-NAT or NAT overlapping networks.
10. Observation: `netstat -rn` shows which prefixes your machine treats as local. `whois 137.97.29.249` at a regional registry will name the holder of a public block; `whois 172.31.0.17` will tell you it is RFC 1918 and nothing more, because nobody holds it.

24.3.6 WORDS - remember these

1. Public address — an address the whole internet can reach — a globally unique address allocated through IANA and a regional registry.
2. Private address — an address meaningful only inside one network — an address from 10.0.0.0/8, 172.16.0.0/12 or 192.168.0.0/16 per RFC 1918, BCP 5.
3. RFC 1918 — the 1996 document reserving private ranges — Best Current Practice 5, "Address Allocation for Private Internets".
4. Bogon — an address or route that should never appear — a martian prefix; traffic or routing announcements from reserved or unallocated space.
5. Ingress filtering — dropping packets with impossible sources at the edge — the practice described in BCP 38 and RFC 2827.

6. Address overlap — two networks using the same private numbers — the merger problem solved by renumbering or twice-NAT.

24.4 The other special ranges

24.4.1 PLAIN - in simple words

1. Private addresses are not the only ones with a special job. Several other blocks are carved out and cannot be used as ordinary addresses.
2. **Loopback**: everything starting with 127. This means "me, this machine". Packets sent there never touch a cable. They turn around inside the software.
3. 127.0.0.1 is the one everybody types. The name for it is `localhost`.
4. **Link-local**: everything starting 169.254. A machine gives itself one of these when it asks for an address and nobody answers.
5. If you ever see 169.254 on your laptop, it means the address request failed. It is a symptom, not a setting.
6. **Shared address space**: everything from 100.64 to 100.127. This block was set aside in 2012 for internet providers to number their customers when they have run out of public addresses.
7. **Multicast**: everything from 224 to 239. These are not one machine. They are a group. One packet, many listeners.
8. **All zeros**, 0.0.0.0, means "no address yet" or "every address here", depending on where it appears.
9. **All ones**, 255.255.255.255, means "everyone on this wire, right now".
10. There are also three blocks reserved purely for writing examples in books and manuals, so that a printed example never accidentally names a real machine.

24.4.2 PLAIN - a picture in your head

1. Think of the reserved words in a language.
2. In English, "here" and "everyone" and "nobody" are not names. You cannot call a child "Everyone" and expect the register to accept it.
3. 127.0.0.1 is the word "here". It always points back at the speaker, whoever is speaking.
4. 255.255.255.255 is the word "everyone in this room". It never leaves the room, because shouting does not travel through walls.
5. 0.0.0.0 is the phrase "no name yet", used by someone who has just walked in and has not been introduced.
6. 169.254.x.x is the name a person invents for themselves when nobody at the door would give them a badge.
7. 192.0.2.x is like the name "John Doe" in a legal form. It is deliberately nobody.

Where this comparison breaks: reserved words in a language are enforced by custom, and you can break the rule and be merely odd. These reserved address blocks are enforced by software. Your operating system will refuse to route 127.x.x.x off the machine even if you order it to, and other people's routers will simply discard the traffic. The rule is mechanical, not social.

24.4.3 PLAIN - a worked example

1. Type this on any machine and it will answer without touching a network:

```
ping -c 2 127.0.0.1
# 64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.045 ms
```

2. A time of about 0.05 milliseconds is not a fast network. It is no network. The packet went down the stack and straight back up.
3. Unplug your Ethernet cable and turn off Wi-Fi, and 127.0.0.1 still works. That is the test that proves it never leaves the machine.

4. Now the failure case. Take a laptop, plug it into a switch with no DHCP server, and wait about twenty seconds.

```
ipconfig getifaddr en0
# 169.254.213.7
```

5. That address was not given to the laptop. The laptop invented it, picked at random from 169.254.0.0/16, after checking nobody else on the wire had it.
6. Two laptops that both do this can talk to each other, and to nothing else. There is no gateway, so nothing routes.
7. Now the carrier case. If your router's outside address is 100.64.9.31, you are not on the internet directly. Your provider has put you behind their own translation. That is carrier-grade NAT, and section 24.9 covers it.
8. The reader's own hops do not show 100.64.x.x. They show RFC 1918 addresses instead. That is a different but equally common carrier choice, and it is discussed honestly in section 24.9.

24.4.4 PLAIN - what is really happening inside

1. For loopback, the operating system's network stack checks the destination before anything reaches a driver. If the destination is on the loopback interface, the packet is handed straight back to the receive path.
2. That is why loopback latency measures the speed of your kernel, not of any wire, and why loopback has an enormous maximum packet size, typically 16384 bytes on macOS instead of the usual 1500.
3. For link-local, the machine picks a random address in 169.254.1.0 to 169.254.254.255, then sends an ARP probe asking "does anyone own this".
4. If somebody answers, it picks another and tries again. If nobody answers after a few tries, it keeps the address.
5. That process is why the address appears about fifteen to twenty seconds after the DHCP attempt fails, not instantly.
6. For multicast, routers and switches maintain group membership. A machine announces "I want group 239.1.1.1" and the network arranges for copies to reach it. One sender, many receivers, one copy per link rather than per receiver.
7. For 255.255.255.255, routers never forward it. It reaches every machine on the local segment and stops. That limit is deliberate; otherwise one packet could reach the whole internet.
8. This is exactly how DHCP starts. A machine with no address at all sends from 0.0.0.0 to 255.255.255.255, because it cannot address the server it has not met yet.

24.4.5 TECHNICAL - the engineer's version

1. The full set of special-purpose IPv4 blocks, each with the document that defines it:

Prefix	Purpose	Defining RFC
0.0.0.0/8	this network	RFC 791
0.0.0.0/32	unspecified source	RFC 1122
10.0.0.0/8	private	RFC 1918
100.64.0.0/10	shared, for CGNAT	RFC 6598
127.0.0.0/8	loopback	RFC 1122
169.254.0.0/16	link-local	RFC 3927
172.16.0.0/12	private	RFC 1918
192.0.0.0/24	IETF protocol use	RFC 6890
192.0.2.0/24	documentation, TEST-NET-1	RFC 5737
192.168.0.0/16	private	RFC 1918
198.18.0.0/15	benchmark testing	RFC 2544

Prefix	Purpose	Defining RFC
198.51.100.0/24	documentation, TEST-NET-2	RFC 5737
203.0.113.0/24	documentation, TEST-NET-3	RFC 5737
224.0.0.0/4	multicast	RFC 1112
240.0.0.0/4	reserved, future use	RFC 1112
255.255.255.255/32	limited broadcast	RFC 919, RFC 8190

2. RFC 1122, "Requirements for Internet Hosts - Communication Layers", October 1989, edited by Robert Braden, is what reserves all of 127.0.0.0/8 for loopback, not just 127.0.0.1. That is 16,777,216 addresses spent on one idea.
3. RFC 3927, "Dynamic Configuration of IPv4 Link-Local Addresses", May 2005, defines 169.254.0.0/16. Apple's implementation is branded Bonjour, and Microsoft's is called APIPA, Automatic Private IP Addressing. Both implement the same RFC.
4. RFC 6598, "IANA-Reserved IPv4 Prefix for Shared Address Space", April 2012, is BCP 153. It reserves 100.64.0.0/10, which is 4,194,304 addresses, from 100.64.0.0 to 100.127.255.255. It exists because carriers were using RFC 1918 space for CGNAT and colliding with customers' own networks.
5. RFC 1112, "Host Extensions for IP Multicasting", August 1989, by Steve Deering, defines 224.0.0.0/4. Assignment inside it is governed by RFC 5771, BCP 51. Notable addresses: 224.0.0.1 all hosts on this subnet, 224.0.0.2 all routers, 224.0.0.5 and 224.0.0.6 for OSPF, 224.0.0.251 for mDNS.
6. RFC 5737, January 2010, reserves the three documentation blocks. This chapter deliberately does not use them, because the reader asked for their own real addresses. Where an address here is invented rather than measured, it is labelled as such.
7. RFC 2544, March 1999, reserves 198.18.0.0/15 for device benchmarking; RFC 6815, November 2012, warns that this traffic must never leave the lab.
8. 240.0.0.0/4 is 268 million addresses sitting unused since 1989. There have been repeated IETF proposals to release it. It has not happened, because every operating system, router and firewall on earth would need updating to accept it, and the gain is roughly one year of global growth.
9. 0.0.0.0/0 as a route means "everything", the default route. 0.0.0.0 as a bind address in a server means "listen on every local address". Same digits, entirely different meaning depending on the field it sits in.
10. Observation: `netstat -rn` on macOS shows the 127.0.0.0/8 route pointing at `lo0` and the `default` route pointing at 192.168.0.1 for the reader.

24.4.6 WORDS - remember these

1. Loopback — the address that means this machine — 127.0.0.0/8 per RFC 1122, handled entirely inside the kernel, never placed on a link.
2. Link-local — the address a machine gives itself when nothing answers — 169.254.0.0/16 per RFC 3927, claimed by ARP probe, no gateway.
3. Shared address space — the block providers use to number customers behind their own NAT — 100.64.0.0/10 per RFC 6598, BCP 153.
4. Multicast — one packet delivered to a group — 224.0.0.0/4 per RFC 1112, with group membership managed by IGMP.
5. Limited broadcast — everyone on this wire and no further — 255.255.255.255/32, never forwarded by a router.
6. Unspecified address — I do not have an address yet — 0.0.0.0 as a source, as used by a DHCP client before it is configured.
7. TEST-NET — addresses reserved for writing examples — 192.0.2.0/24, 198.51.100.0/24 and 203.0.113.0/24 per RFC 5737.

24.5 Address exhaustion: why 4.29 billion was not enough

24.5.1 PLAIN - in simple words

1. There are 4,294,967,296 possible IPv4 addresses. There are more than 8 billion people, and far more than 8 billion devices.
2. So even in the best case the numbers do not work. But the shortage arrived much earlier than the raw count suggests.
3. It arrived early because of how the addresses were handed out, not because they were all in use.
4. In the beginning, addresses were given out in three fixed sizes only: about 16 million, about 65 thousand, or 256.
5. There was nothing in between. If your university needed 400 addresses, 256 was too few, so you got 65,536 and wasted 65,136 of them.
6. Big early organizations were handed 16 million addresses each, often for a few thousand machines.
7. By 1992 people could see the end coming. Not just the end of addresses, but the end of the routers, which were running out of memory to hold all the separate routes.
8. In 1993 the fixed sizes were abolished and replaced by the slash notation of section 24.2. That let blocks be any size, which stopped most of the waste.
9. Private addresses plus NAT arrived soon after, which let one public address serve a whole household or office.
10. Those two changes bought about twenty extra years. The free pool of unused addresses still ran dry, starting in 2011.

24.5.2 PLAIN - a picture in your head

1. Imagine a city where land is sold only in three plot sizes: a whole district, a whole street, or a single house.
2. A family needing two houses must buy a whole street, and leave the rest empty, because there is no plot of two.
3. A school needing thirty houses must also buy a street. So does a factory needing sixty.
4. Early buyers, arriving when land felt endless, took whole districts for a handful of buildings.
5. The city fills up not because it is full of people, but because it is full of fences around empty ground.
6. In 1993 the city changed the rules: you can now buy any number of plots you actually need. That helped enormously, but the districts already sold stayed sold.

Where this comparison breaks: land can be compulsorily bought back and redeveloped. IPv4 blocks mostly cannot. Some early holders have sold blocks on an open market, and some space has been returned, but there is no authority that can take an allocation back by force. The fences from 1985 are still standing.

24.5.3 PLAIN - a worked example

1. Here is the old class system, the one abolished in 1993.

Class	First bits	Prefix	Hosts each
A	0	/8	16,777,214
B	10	/16	65,534
C	110	/24	254
D	1110	multicast	not applicable
E	1111	reserved	not applicable

2. The class was decided by the leading bits of the address itself, so you could tell the network size by looking at the first number.
3. First number 1 to 126 meant class A. 128 to 191 meant class B. 192 to 223 meant class C.

4. Check the reader's own addresses against that. 192.168.0.1 begins with 192, so it sat in the old class C range. 137.97.29.249 begins with 137, so class B. 20.207.73.82 begins with 20, so class A.
5. That last one is real history, not trivia. The 20.0.0.0/8 block, all 16.7 million addresses, was allocated in the 1980s to a single company. Microsoft later held it, and today it fronts services including the GitHub endpoint the reader was trying to reach.
6. Now the waste. Suppose a company in 1990 needs 400 addresses.
7. A class C gives 254. Not enough.
8. A class B gives 65,534. So they get a class B, and 65,134 addresses sit idle forever.
9. With CIDR the same company gets a /23, which is 512 addresses, 510 usable. Waste drops from 65,134 to 110.
10. Multiply that saving across every allocation made after 1993 and you have the reason the internet did not stop growing in 1996.

24.5.4 PLAIN – what is really happening inside

1. Two separate crises were happening at once, and it helps to keep them apart.
2. The first is **address exhaustion**: running out of numbers to give people.
3. The second is **routing table growth**: the routers in the middle of the internet were running out of memory to store one entry per network.
4. Class B allocation was making both worse. Every class B handed out was one more route every core router had to remember.
5. CIDR fixed both. Smaller allocations meant less waste, and blocks handed to one provider could be advertised as one summary route instead of hundreds.
6. NAT then attacked the first problem from the other end: instead of finding more addresses, it made one address serve many machines.
7. The IANA free pool is the top of a three-level tree. IANA holds unallocated space, gives large blocks to five regional registries, and those registries give blocks to providers and companies.
8. When people say the internet ran out in 2011, they mean the top level ran dry. The registries kept issuing from their own stock for years afterwards, under increasingly strict rules.
9. Today, in most regions, new IPv4 space is obtained by buying it from an existing holder on a transfer market, not by asking a registry.

24.5.5 TECHNICAL – the engineer's version

1. Total IPv4 space is 2 to the power 32, exactly 4,294,967,296. Subtracting the special blocks of section 24.4, roughly 3.7 billion are usable as ordinary public unicast addresses.
2. The exhaustion timeline, with verified dates:

Date	Event
31 January 2011	IANA gave last 2 free /8s
3 February 2011	NRO declared pool depleted
15 April 2011	APNIC hit final /8 policy
September 2012	RIPE NCC hit final /8
10 June 2014	LACNIC exhaustion
24 September 2015	ARIN exhaustion
2017	AFRINIC entered phase 1
25 November 2019	RIPE NCC fully out

3. The precision matters. On 31 January 2011 IANA allocated 39.0.0.0/8 and 106.0.0.0/8 to APNIC, which triggered the "final five" policy under which the five remaining /8s were distributed one to each registry. The Number Resource Organization made the formal announcement on 3 February 2011. So both dates are correct, for different events.

4. APNIC, serving Asia-Pacific including India, reached its final /8 first, on 15 April 2011, because demand there was highest. That is directly relevant to the reader: an Indian ISP has been operating under severe IPv4 scarcity for over a decade, which is the background to section 24.9.
5. CIDR was specified in RFC 1518 and RFC 1519, both September 1993, and deployed through 1994. RFC 4632, August 2006, BCP 122, is the current specification.
6. Classful addressing was formally deprecated by RFC 4632. Some textbooks and certification syllabuses still teach classes; treat them as history that explains why blocks look the way they do, not as current practice.
7. The measures that bought time, in rough order of impact: CIDR from 1993, NAT with RFC 1918 from 1994 to 1996, DHCP address recycling from RFC 2131 in 1997, and carrier-grade NAT from around 2010.
8. Global BGP routing table size is the counter-measure to watch. It passed roughly 100,000 routes around 2000 and is now well past 900,000 IPv4 routes. Aggregation slowed the growth; it did not stop it.
9. Transfer markets now set a real price for IPv4. Reported per-address prices rose from a few dollars in 2011 into the mid-tens of dollars by the mid-2020s, with real volatility. Treat any single quoted figure as approximate and dated.
10. Amazon Web Services began charging for every public IPv4 address on 1 February 2024, at 0.005 US dollars per address per hour. That is the clearest signal that IPv4 became a priced commodity rather than a free resource.

24.5.6 WORDS - remember these

1. Address exhaustion — running out of numbers to hand out — depletion of unallocated IPv4 unicast space at IANA and the regional registries.
2. Classful addressing — the old fixed sizes A, B and C — the pre-1993 scheme where the leading bits determined the prefix length; deprecated by RFC 4632.
3. CIDR — letting blocks be any size — Classless Inter-Domain Routing, RFC 1518 and RFC 1519 in 1993, now RFC 4632.
4. IANA — the body holding the master pool — Internet Assigned Numbers Authority, operated under Public Technical Identifiers within ICANN.
5. RIR — a regional address registry — one of AFRINIC, APNIC, ARIN, LACNIC and RIPE NCC.
6. Free pool — unallocated addresses still available — the stock a registry can issue without a transfer.
7. Transfer market — buying addresses from an existing holder — registry- approved reassignment of allocated space, now the main source of IPv4.

24.6 NAT, explained completely

24.6.1 PLAIN - in simple words

1. You have one public address, from your provider. You have a laptop, a phone, a television and a printer. All four want the internet at once.
2. NAT is the trick that lets all four share the one address.
3. NAT stands for Network Address Translation. The router rewrites addresses as packets pass through it.
4. Going out, the router takes your private source address and replaces it with its own public one. The far end never sees your real address.
5. Coming back, the router does the reverse. It puts your private address back before delivering the reply to you.
6. To do the reverse it must remember which reply belongs to which machine. So it keeps a table.
7. The table has one row per conversation. It is created when you start a conversation and deleted when the conversation ends or goes quiet.

8. The key trick is that the router also rewrites the port number, not just the address. That is what allows many machines to share one address at once.
9. Ports are section 24.7. For now, treat a port as a numbered slot on the router, one per conversation.
10. Nothing on the internet knows any of this happened. To the far end, it looks like one machine with one address doing a lot of things.

24.6.2 PLAIN – a picture in your head

1. Think of a hotel with one street address and one post room.
2. A guest in room 214 writes a letter and asks for a reply.
3. The post room will not put "Room 214" on the envelope. It writes the hotel's street address, and adds a reference number: "reply to reference 51022".
4. It notes in a ledger: reference 51022 belongs to room 214, and the letter went to this particular company.
5. The reply arrives addressed to the hotel, quoting reference 51022.
6. The post room looks up 51022 in the ledger, sees room 214, writes that on the envelope, and puts it in the guest's pigeonhole.
7. A thousand guests can write letters at once, because each gets a different reference number.
8. If a letter arrives quoting a reference that is not in the ledger, the post room has no idea what to do with it. It goes in the bin.

Where this comparison breaks: a hotel post room can be told in advance "any letter for the conference goes to room 214", which is what port forwarding is. But without that instruction, unexpected post is genuinely undeliverable, not merely inconvenient. Also, the hotel ledger is permanent; a NAT table entry is deleted after a timeout, typically a few minutes for a quiet UDP conversation, which is why idle connections die and why applications send keep-alives.

24.6.3 PLAIN – a worked example

1. Here is the reader's own situation, with the one value we have to invent clearly marked.
2. Known from the reader's session: the router is 192.168.0.1, and github.com resolved to 20.207.73.82, and the connection was to port 443, which is HTTPS.
3. Not recorded in the session: the laptop's own address and the router's public address. We will use 192.168.0.24 for the laptop, which is a plausible value in the reader's 192.168.0.0/24 network, and we mark the public address as an example.
4. The laptop opens a connection. The operating system picks a random unused local port, say 52344.
5. The packet leaving the laptop looks like this:

```
source      192.168.0.24:52344
destination 20.207.73.82:443
protocol    TCP
```

6. It reaches 192.168.0.1. The router rewrites the source and writes a row in its table.
7. The packet leaving the router looks like this:

```
source      203.0.113.45:61001   (example public address)
destination 20.207.73.82:443
protocol    TCP
```

8. And the table row it created:

Internal	External	Destination
192.168.0.24:52344	203.0.113.45:61001	20.207.73.82:443

9. Now the full table with three devices in the flat, all talking at once. This is what a real home router holds.

Internal addr:port	External port	Dest addr:port	Proto
192.168.0.24:52344	61001	20.207.73.82:443	TCP
192.168.0.24:52345	61002	20.207.73.82:443	TCP
192.168.0.24:52346	61003	140.82.121.4:22	TCP
192.168.0.31:49780	61004	1.1.1.1:53	UDP
192.168.0.31:51201	61005	20.207.73.82:443	TCP
192.168.0.57:60110	61006	17.253.144.10:123	UDP

10. Read row 1 and row 5 together. Two different machines, both talking to 20.207.73.82 on port 443, both leaving as the same public address. They are told apart only by the external port, 61001 versus 61005.
11. A reply arrives at the router addressed to 203.0.113.45 port 61005. The router looks up 61005, finds 192.168.0.31:51201, rewrites the destination, and forwards it inside.
12. Now the crucial negative case. A packet arrives from the internet addressed to 203.0.113.45 port 8080, and no row mentions 8080. The router has no idea which of the three machines it is for. It drops it.
13. That single drop is the whole of section 24.8.

24.6.4 PLAIN - what is really happening inside

1. Rewriting the address is not just a copy. The router must also fix the checksums.
2. A checksum is a small number computed from the packet contents, used to catch corruption. Change the address and the checksum no longer matches.
3. So the router recomputes the IP header checksum, and also the TCP or UDP checksum, because those are computed over the addresses as well.
4. Good implementations do this incrementally: they adjust the old checksum by the difference rather than recomputing from scratch, which is far faster.
5. The table entry is created by the first packet of a conversation and removed on a timer, or when the connection closes cleanly.
6. For TCP, the router can watch the connection state. It sees the opening handshake and the closing exchange, so it knows when a row is finished.
7. For UDP, there is no handshake and no close. The router can only use a timer, typically 30 to 300 seconds of silence.
8. This is why a video call freezes if you pause it for a few minutes, and why chat applications send a tiny packet every 20 to 30 seconds for no reason other than keeping the NAT row alive.
9. The table lives in memory and is finite. Cheap home routers hold a few thousand rows. Fill it and new connections simply fail, often with no useful error.
10. The router also has to be careful about port collisions. If two internal machines happen to pick the same source port to the same destination, the router must allocate different external ports, which it does automatically.

24.6.5 TECHNICAL - the engineer's version

1. NAT was first described in RFC 1631, "The IP Network Address Translator", May 1994, by Kjeld Egevang and Paul Francis. It was explicitly proposed as a short-term measure.
2. Terminology was standardized in RFC 2663, August 1999. The current description of the common case is RFC 3022, "Traditional IP Network Address Translator", January 2001, which obsoletes RFC 1631.
3. RFC 3022 separates two things. **Basic NAT** translates addresses only. **NAPT**, Network Address Port Translation, translates address and transport identifier together. Home routers do NAPT.
4. The industry name for NAPT is **PAT**, Port Address Translation, in Cisco documentation, or **NAT overload**. Linux calls it **masquerade**. All four words describe the same mechanism.
5. The four kinds you must be able to name:

Kind	What it rewrites	Typical use
Source NAT	source addr on egress	home router outbound
Destination NAT	dest addr on ingress	port forwarding
Static NAT	one to one, both ways	a public server
PAT / NAPT	addr plus port, many to one	every home router

6. The state a NAT keeps per flow is keyed on the 5-tuple: protocol, source address, source port, destination address, destination port. That is what makes rows unique even when several fields repeat.
7. NAT behaviour is classified by RFC 4787, BCP 127, January 2007, for UDP, and RFC 5382 for TCP. The mapping behaviours are endpoint-independent, address-dependent, and address-and-port-dependent. Endpoint-independent mapping is required by BCP 127 because it is the only behaviour that makes peer-to-peer hole punching reliable.
8. The older names cone NAT, restricted cone, port-restricted cone and symmetric NAT come from RFC 3489, the original STUN document from 2003. That classification was found too coarse and was deliberately replaced by the RFC 4787 vocabulary. You will still see the old names in game-console diagnostics, where "NAT type 3" means a restrictive NAT.
9. Default timeouts, as a guide rather than a standard: established TCP flows are often held 2 to 24 hours, TCP in the opening handshake around 4 minutes, UDP 30 to 180 seconds, ICMP around 30 seconds. RFC 5382 requires at least 2 hours 4 minutes for established TCP; RFC 4787 requires at least 2 minutes for UDP.
10. Observation on Linux: `conntrack -L` lists live NAT and connection tracking rows, and `cat /proc/sys/net/netfilter/nf_conntrack_max` gives the table size. On macOS the relevant tool is `pfctl -s state`. On a home router the table is usually visible in the web interface under a name like "NAT sessions" or "connection list".

```
# Linux router: see the live translation table
sudo conntrack -L | head -3
# tcp 6 431999 ESTABLISHED src=192.168.0.24 dst=20.207.73.82
#   sport=52344 dport=443 src=20.207.73.82 dst=203.0.113.45
#   sport=443 dport=61001 [ASSURED]
```

24.6.6 WORDS - remember these

1. NAT — rewriting addresses as packets cross a boundary — Network Address Translation, RFC 2663 and RFC 3022.
2. NAPT or PAT — rewriting address and port so many can share one — Network Address Port Translation, the mechanism in every home router.
3. Source NAT — changing where a packet says it came from — SNAT, applied on egress, the outbound direction.
4. Destination NAT — changing where a packet is going — DNAT, applied on ingress; port forwarding is a DNAT rule.
5. Static NAT — a fixed one-to-one mapping in both directions — a bidirectional NAT binding, used to publish a server on a dedicated public address.
6. NAT table — the router's memory of who is talking to whom — the translation state table, keyed on the 5-tuple.
7. 5-tuple — the five facts that identify one conversation — protocol, source address, source port, destination address, destination port.
8. Masquerade — the Linux word for many-to-one source NAT — an iptables or nftables target that uses the egress interface's current address.

24.7 Ports: the trick that makes NAT work

24.7.1 PLAIN - in simple words

1. An address gets a packet to the right machine. A **port** gets it to the right conversation on that machine.
2. A port is just a number. It is 16 bits, so it runs from 0 to 65535.
3. It is not a physical thing. There is no socket on the back of the computer labelled 443. It is a number written in the packet header.
4. Your machine may be running a web browser, a mail program, a music stream and a code editor, all talking at once. Every one of those conversations has its own port number.
5. When data arrives, the operating system reads the port and hands the data to the program that claimed that number.
6. A server claims a fixed, well-known port so clients can find it. Web servers with encryption claim 443. That is why the reader's `curl` said port 443.
7. A client does not need a fixed number. It grabs any free one, uses it for that one conversation, and releases it afterwards.
8. Those temporary client numbers are called **ephemeral** ports, meaning short-lived.
9. NAT depends completely on this. Because the router can change the port as well as the address, it can give every conversation a unique number even when they all share one address.
10. Without ports, one public address could carry exactly one conversation at a time, and NAT would be useless.

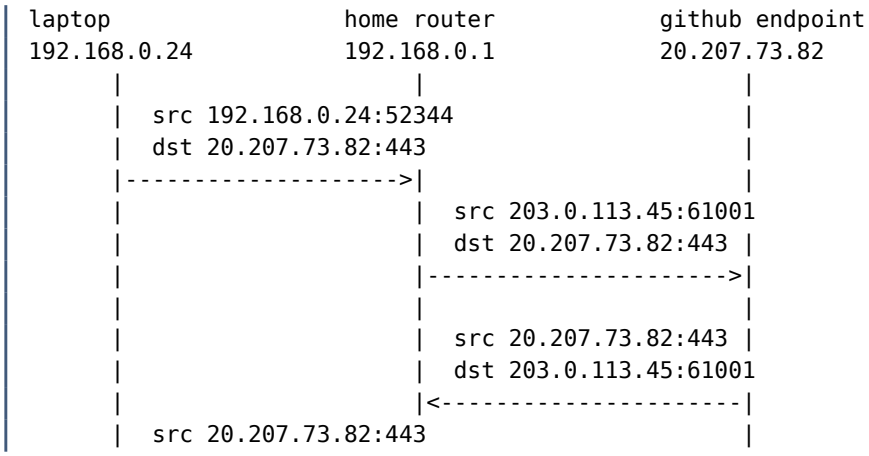
24.7.2 PLAIN - a picture in your head

1. Think of a large company with one street address and many departments.
2. The street address gets the post to the building. "Attention: Accounts" gets it to the right desk inside.
3. Some departments have famous, published names. Everyone knows to write "Attention: Sales" without asking. Those are the well-known ports.
4. When Sales writes out to a supplier, it says "reply to case 52344". That case number is invented for this one exchange and forgotten afterwards. That is an ephemeral port.
5. Two letters can go to the same supplier on the same day from the same building, and come back correctly, because the case numbers differ.

Where this comparison breaks: a company can invent unlimited case numbers. A machine has exactly 65,536 port numbers per protocol per address, and no more. That hard ceiling is a real engineering limit that busy servers and carrier NAT boxes genuinely run into, and it is the subject of the technical block below.

24.7.3 PLAIN - a worked example

1. Here is the full picture of the reader's connection attempt, showing address and port at every stage.



```
|  dst 192.168.0.24:52344                    |
|<--------------------|                      |
```

2. Note what stayed the same all the way through: the destination port 443. The server's port is never rewritten by your router.
3. Note what changed: the source address and the source port, both rewritten outbound and both restored inbound.
4. Here are the port ranges and what each is for.

Range	Name	Who uses it
0 to 1023	System, well-known	servers, needs privilege
1024 to 49151	User, registered	named services
49152 to 65535	Dynamic, ephemeral	clients, never assigned

5. Some well-known ports worth memorizing, several of which appear in the reader's own session:

Port	Service	Note
22	SSH	git over SSH
53	DNS	the reader used 1.1.1.1
80	HTTP	unencrypted web
123	NTP	clock sync
443	HTTPS	git over HTTPS, curl
993	IMAPS	encrypted mail fetch
3306	MySQL	database
5432	PostgreSQL	database

6. This table explains something from the reader's own session directly. Git over SSH uses port 22. Git over HTTPS uses port 443. They are different conversations to possibly different machines, so one can fail while the other works.

24.7.4 PLAIN - what is really happening inside

1. The port numbers are not in the IP header. They are in the next header inside, the TCP or UDP header.
2. In both TCP and UDP, the source port is the first two bytes and the destination port is the next two bytes. They are the very first thing in that header, which makes them cheap for hardware to read.
3. When your program asks to open a connection, it usually does not choose a port. It asks the operating system for any free one.
4. The operating system keeps a record of which ports are in use, picks a free one from its ephemeral range, and marks it taken.
5. Modern systems pick randomly within the range rather than counting upwards. That is a deliberate security change, because predictable ports made certain attacks easier.
6. The combination that must be unique is not the port alone. It is the whole 5-tuple: protocol, both addresses, both ports.
7. That is why a busy web server can have 200,000 connections open while only listening on one port: every client is a different address or a different client port.
8. A server binding to a port below 1024 traditionally needs administrator rights on Unix systems. That is a convention from the 1980s designed to stop ordinary users impersonating system services.
9. When the router rewrites a port, it must pick one that is not already used for another conversation to the same destination. Its allocator does that check on every new flow.

24.7.5 TECHNICAL - the engineer's version

1. Port numbers are 16-bit unsigned integers, 0 to 65535, defined separately for each transport protocol. TCP port 443 and UDP port 443 are different registry entries and different listening sockets.
2. The three ranges are defined in RFC 6335, August 2011, which is BCP 165. The official names are System Ports, User Ports and Dynamic Ports; the informal names are Well Known, Registered and Private or Ephemeral.
3. Operating systems do not agree on the ephemeral range, and this is an implementation detail, not a standard:

System	Default ephemeral range
macOS and FreeBSD	49152 to 65535
Linux	32768 to 60999
Windows Vista onward	49152 to 65535
Windows XP and earlier	1025 to 5000

4. On Linux the range is readable and writable at `/proc/sys/net/ipv4/ip_local_port_range`. On macOS it is `sysctl net.inet.ip.portrange.first` and `.last`.
5. Port 0 is reserved and must not be used on the wire. Binding to port 0 in a socket API is a request for the kernel to allocate an ephemeral port, which is a different meaning of the same number.
6. The theoretical connection limit. For a single client address talking to a single server address and port, the only free field is the client port, so the ceiling is 65,536 minus reserved values, roughly 64,500 simultaneous connections. If the operating system uses the standard ephemeral range only, the practical ceiling is about 16,384 on macOS and about 28,000 on Linux.
7. For NAT, the same arithmetic applies to the public address. One public address can support roughly 64,500 simultaneous flows to any one destination endpoint, but many times that across different destinations, because the destination fields also vary.
8. In carrier practice, providers plan on a sharing ratio rather than the theoretical limit. Published deployments have used anywhere from 4 to over 100 subscribers per public address, with per-subscriber port quotas of a few hundred to a few thousand. RFC 6888 requires the ability to set such a quota precisely so one subscriber cannot exhaust the pool.
9. Port exhaustion is a real production failure mode. Its symptoms are new connections failing while existing ones work, and on Linux a rising count of sockets in `TIME_WAIT`, which holds a port for 2 times the maximum segment lifetime, usually 60 seconds total.
10. Observation commands:

```
lsof -i -nP | head           # macOS: which program owns a port
netstat -an | grep 443          # all conversations on port 443
sysctl net.inet.ip.portrange.first net.inet.ip.portrange.last
ss -tan state time-wait | wc -l # Linux: count TIME_WAIT sockets
```

24.7.6 WORDS - remember these

1. Port — a number identifying one conversation on a machine — a 16-bit field in the TCP or UDP header, first four bytes of that header.
2. Well-known port — a published number a server listens on — System Ports 0 to 1023, assigned by IANA under RFC 6335, BCP 165.
3. Ephemeral port — a temporary number a client borrows — Dynamic Ports 49152 to 65535 by IANA, with operating system defaults that differ.
4. Socket — one end of a conversation — the pairing of an address and a port, plus a protocol, exposed as a file descriptor by the operating system.
5. Port exhaustion — running out of conversation numbers — depletion of the available ephemeral or NAT external port pool, causing new connections to fail while old ones survive.

6. TIME_WAIT — a closed connection still holding its number — the TCP state that retains the 4-tuple for twice the maximum segment lifetime.

24.8 What NAT breaks

24.8.1 PLAIN - in simple words

1. NAT works beautifully in one direction and badly in the other.
2. Outward is fine. You start the conversation, the router makes a table row, replies find their way home.
3. Inward is the problem. Nobody outside can start a conversation with you.
4. When an unexpected packet arrives, the router looks in its table, finds nothing, and has no way to guess which of your devices it was for. So it drops it.
5. This means you cannot run a server at home that other people reach, unless you configure the router specially.
6. That special configuration is called **port forwarding**: a permanent rule saying "anything arriving on port 8080 goes to 192.168.0.24".
7. It also means two people both behind NAT cannot simply call each other. Neither can start the conversation, because both are unreachable from outside.
8. Video calls, online games and file sharing all hit this. They solve it with helper servers and a trick called hole punching.
9. Some older protocols carry addresses inside their own messages. NAT rewrites the header but not the message, so those protocols break in confusing ways.
10. The internet was originally designed so that any machine could talk to any other machine directly. NAT ended that. It is a workaround that became permanent.

24.8.2 PLAIN - a picture in your head

1. Go back to the hotel post room from section 24.6.
2. A guest writes out and gets a reply. That works, because the ledger has the reference number.
3. Now imagine somebody outside wants to write to the guest first, without being asked. They know the hotel address. They do not know the guest is there, and they certainly do not know any reference number.
4. Their letter arrives with nothing useful on it. The post room genuinely cannot deliver it. There are four hundred rooms.
5. Worse: two guests in two different hotels want to write to each other. Each can write out. Neither can be written to. Their letters cross in the post and both land in bins.
6. The only fix is a third party. Both guests write to a shared friend who already has a public, reachable address. The friend tells each one the other's hotel address and reference number, at the same moment, so both send first and both ledgers get their rows.

Where this comparison breaks: this is where the comparison stops being a comparison and becomes literally true. The shared friend is a STUN server, the simultaneous sending is hole punching, and if it fails, the friend physically relays every letter, which is a TURN server. The technical block names them.

24.8.3 PLAIN - a worked example

1. Say the reader wants to run a small web server on the laptop at 192.168.0.24, listening on port 8080, and let a friend see it.
2. The friend types the public address into a browser. The packet arrives at the router addressed to port 8080.
3. The router checks its table. There is no row for port 8080. It drops the packet. The friend sees a connection timeout with no error message. Silence.
4. The fix is a port forwarding rule on the router:

Public port	Internal address	Internal port
8080	192.168.0.24	8080

5. That rule is a destination NAT entry, permanently installed rather than created by traffic. Now the packet is rewritten and delivered.
6. But note what you have just done: you have permanently allocated one public port to one machine. You cannot forward port 8080 to two laptops.
7. Now the peer-to-peer case. Two friends both behind home NAT want a video call.
8. Each contacts a public helper server and asks "what address and port do you see me as". The helper answers with the outside view, which is the router's public address and the chosen external port.
9. Each sends that answer to the other, through a signalling channel that both can already reach.
10. Then both send a packet to the other at the same moment. Each outgoing packet creates a table row in its own router. The incoming packet then matches a row that already exists, so it is delivered.
11. That is hole punching. Both sides must send first, which is why it needs careful timing and a coordinator.
12. If the routers are too restrictive for that to work, both sides give up and send everything through a relay server instead. That works always, but costs the relay operator bandwidth, and adds delay.

24.8.4 PLAIN – what is really happening inside

1. The deep problem is that NAT breaks an assumption the internet was built on.
2. The original assumption is that an address means the same thing everywhere. The address you have is the address others use to reach you.
3. Under NAT that is false. You have one address; the world sees another. There is no way for your machine to know its own public address without asking somebody outside.
4. Any protocol that puts an address inside its own message, rather than only in the header, is therefore broken by NAT.
5. Old-style FTP does exactly that. In active mode the client sends its own address and port as text inside a command, and asks the server to connect back. Under NAT the address it sends is private and useless.
6. Telephony signalling has the same shape. It negotiates media addresses inside the message body, and those addresses are private.
7. The workaround is an **ALG**, an Application Layer Gateway: the router inspects the payload, finds the address written in text, and rewrites that too.
8. That is a genuinely ugly solution. The router now has to understand every application protocol, and it breaks when the protocol changes or is encrypted.
9. Encryption killed the ALG approach for good. If the payload is encrypted, no middle box can rewrite it. This is one reason modern protocols never put addresses in the payload.
10. There is a second cost that is easy to miss. NAT makes the network stateful. A router that only forwards can be rebooted or replaced and traffic continues. A router holding NAT state loses every conversation when it restarts.

24.8.5 TECHNICAL – the engineer's version

1. The design principle NAT violates is stated in "End-to-End Arguments in System Design" by Jerome Saltzer, David Reed and David Clark, presented in 1981 and published in ACM Transactions on Computer Systems in November 1984. The argument is that functions should live at the endpoints, and the network should stay simple.
2. NAT puts per-conversation state in the middle of the network. That is precisely what the end-to-end argument says not to do.
3. The NAT traversal stack, with current documents:

Protocol	Job	Current RFC
STUN	discover your outside view	RFC 8489, Feb 2020
TURN	relay when nothing else works	RFC 8656, Feb 2020
ICE	try candidates, pick one	RFC 8445, July 2018

4. RFC 8489 obsoletes RFC 5389 from October 2008, which in turn replaced RFC 3489 from March 2003. RFC 8656 obsoletes RFC 5766 from February 2010. RFC 8445 obsoletes RFC 5245 from 2010.
5. ICE gathers three kinds of candidate address: host candidates from local interfaces, server-reflexive candidates learned from STUN, and relayed candidates allocated on a TURN server. It then runs connectivity checks in priority order and uses the best pair that works.
6. In practice a WebRTC video call in a browser is running exactly this machinery. Roughly speaking, most calls succeed with a direct or server-reflexive path, and a minority fall back to TURN relay. The exact fallback rate depends heavily on the network population being measured, so treat any single published percentage with caution.
7. Protocols historically needing an ALG: FTP in active mode, where RFC 959's `PORT` command carries an address as ASCII text; SIP, where addresses appear in the SDP body; H.323; and some peer-to-peer file sharing protocols.
8. IPsec is a specific casualty. Authentication Header protects the IP header including addresses, so any NAT rewrite invalidates it. The workaround is NAT Traversal, RFC 3947 and RFC 3948, January 2005, which wraps ESP inside UDP on port 4500.
9. UPnP IGD and NAT-PMP, and their successor Port Control Protocol in RFC 6887, April 2013, let an internal device ask the router to open a mapping. They work, and they are also a real security concern, since any program on the network can open a hole without asking a human.
10. The honest verdict, stated plainly: NAT is a workaround. RFC 1631 said so in 1994, calling it a short-term solution while a longer-term one was developed. The longer-term solution was IPv6. Thirty years later the short-term measure is still carrying most of the world's traffic. Experts do disagree about whether that is a tragedy or a pragmatic success; the engineering costs listed above are not disputed by either side.

24.8.6 WORDS - remember these

1. Port forwarding — a permanent rule sending one outside port to one inside machine — a static destination NAT entry.
2. Hole punching — both sides sending first so both NAT tables get a row — the simultaneous-open technique used by ICE.
3. STUN — asking a public server what address it sees you as — Session Traversal Utilities for NAT, RFC 8489.
4. TURN — routing everything through a public relay when nothing else works — Traversal Using Relays around NAT, RFC 8656.
5. ICE — trying every possible path and using the best one — Interactive Connectivity Establishment, RFC 8445.
6. ALG — a router that reads and rewrites inside the message — Application Layer Gateway, needed by FTP and SIP, defeated by encryption.
7. End-to-end principle — keep the clever parts at the edges — the design argument of Saltzer, Reed and Clark, 1981 and 1984.
8. Stateful — the box in the middle remembers your conversation — connection state held outside the endpoints, lost on restart.

24.9 Carrier-grade NAT

24.9.1 PLAIN - in simple words

1. You already know that your home router shares one public address between all your devices.
2. Carrier-grade NAT is the same idea, one level up. Your internet provider shares one public address between many customers.
3. They do it for exactly the same reason you do. They have run out of public addresses.
4. So instead of giving your router a public address, they give it a private one, and translate again at a big machine in their own network.
5. Your traffic is now translated twice: once by your router, once by theirs. The usual name for that is double NAT.
6. From your side, nothing looks different for ordinary browsing. Pages load. Video plays.
7. From your side, several things become impossible or unreliable. You cannot accept incoming connections at all, not even with port forwarding, because the outer translation is not yours to configure.
8. You also share a public address with strangers. If one of them misbehaves and a website blocks that address, you are blocked too.
9. Diagnosing faults becomes harder, because there is a machine in the path you cannot see and cannot ask questions of.
10. The reader's own traceroute shows private addresses for five of the first six hops. That is the fingerprint of a provider running a large private core, and it is consistent with carrier-grade NAT.

24.9.2 PLAIN - a picture in your head

1. Go back to the hotel one final time, and add a floor.
2. Your flat is a hotel with extensions. Fine.
3. Now the whole street is served by one switchboard, and your hotel's "main number" turns out not to be a real public number at all. It is an extension on the street switchboard.
4. Outsiders dialling the street switchboard reach an operator who has no idea which hotel, let alone which room, they want.
5. You can still call out. You always could. Calling in is now doubly impossible.
6. And if a neighbour on your street makes nuisance calls, the number that gets barred is the street's number, which is also yours.

Where this comparison breaks: a street switchboard would be an obvious, visible thing. Carrier-grade NAT is invisible to customers. Providers rarely announce it, there is no light on your router for it, and the only way to find out is the test in section 24.13.

24.9.3 PLAIN - a worked example

1. Here is the reader's own first six hops again, exactly as recorded.

```
1  192.168.0.1
2  172.31.0.17
3  137.97.29.249
4  172.26.22.235
5  172.16.18.33 / 172.26.14.75 / 172.26.22.234
6  172.26.14.75 / 172.16.18.33
```

2. Hop 1 is the reader's own router, inside the flat, 192.168.0.1, private.
3. Hop 2 is 172.31.0.17. Second octet 31, inside the 16 to 31 range of 172.16.0.0/12. Private. This is the provider's first router, not the reader's.
4. Hop 3 is 137.97.29.249. Not in any private block. Public.
5. Hops 4, 5 and 6 are all in 172.16.0.0/12. Private again.
6. So the pattern is: private, private, public, private, private, private.

7. That is not a mistake and not a loop. It is what a provider network looks like when most internal links are numbered from private space, but some equipment has a public address on an interface.
8. Now the honest part, and it matters more than anything else in this section.
9. A traceroute shows the address of the interface that generated each reply. It does not show what address your packet was carrying at that point.
10. So the private addresses at hops 2, 4, 5 and 6 prove one thing: the provider numbers those internal links from RFC 1918 space.
11. They do not, on their own, prove that your traffic was translated to a shared public address. That is strongly suggested, not proven.
12. What would prove it is a two-line test, given in full in section 24.13: compare the address on your router's outside interface with the address a public website reports. If they differ, there is a translation between you and the internet that is not yours.

24.9.4 PLAIN - what is really happening inside

1. A carrier-grade NAT is a large, expensive box, or a software function on a large router, sitting between the provider's access network and its public edge.
2. It holds the same kind of table as your home router, but with millions of rows instead of thousands.
3. Each customer is allocated a slice of the port space on a shared public address. A common design gives each subscriber a block of a few thousand ports.
4. Giving each subscriber a contiguous block, rather than picking ports one at a time, is a deliberate choice. It makes logging vastly cheaper: one log line per subscriber per session instead of one per connection.
5. Logging matters because law enforcement requests name an address, a port and a time, and the provider must be able to say which customer that was.
6. What your packet actually experiences, going out:

```
laptop 192.168.0.24
  -> home router, source becomes the WAN address
  -> provider access network (private numbering)
  -> carrier NAT, source becomes a shared public address
  -> provider public edge
  -> the rest of the internet
```

7. Coming back, the reverse happens, twice, and both tables must still hold the row. If either times out, the connection dies.
8. Because the outer table is not yours, you cannot create a permanent inbound rule in it. Port forwarding on your own router still works for traffic that reaches your router, but nothing from the internet reaches your router unasked in the first place.
9. Where hop 3 fits: a public address on a provider router interface tells you that at least that interface is globally addressable. It marks a place where the provider's equipment touches public space. It does not by itself mark the exact point where your source address was rewritten, and it would be overclaiming to say that it does.

24.9.5 TECHNICAL - the engineer's version

1. Carrier-grade NAT is also called CGN, large-scale NAT or LSN. The requirements document is RFC 6888, "Common Requirements for Carrier-Grade NATs", April 2013, which is BCP 127.
2. RFC 6598, April 2012, BCP 153, reserves 100.64.0.0/10 as Shared Address Space for exactly this purpose: numbering the link between the CGN and customer premises equipment. Its existence is a direct admission that carriers were already using RFC 1918 space and colliding with customers' home networks.
3. RFC 6264, June 2011, describes the incremental CGN approach for IPv6 transition. RFC 6598's own justification is the collision problem.
4. Key RFC 6888 requirements worth knowing: paired address pooling, meaning all sessions from one internal address must use the same external address; a configurable per-subscriber port limit; and a minimum 120-second delay before an external port is reused.
5. Deployment forms include NAT444, where there are three address realms and two translations, and DS-Lite, RFC 6333, August 2011, which tunnels IPv4 over IPv6 to a carrier translator called an AFTR. MAP-E and MAP-T, RFC 7597 and RFC 7599, both July 2015, are stateless alternatives.
6. What CGNAT costs the customer:

Effect	Consequence
No unsolicited inbound	cannot host anything
Port forwarding useless	your rule stops at you
Shared reputation	blocked for a stranger
CAPTCHA and rate limits	per-IP limits hit sooner
Harder diagnosis	an invisible stateful hop
Game and call quality	strict NAT, more TURN relaying
Geolocation errors	mapped to the wrong city

7. The shared reputation problem is the one users notice most. Per-address rate limiting, abuse blocklists and anti-fraud scoring all assume one address is roughly one customer. Under CGNAT that assumption is simply false, and it can put hundreds of households behind one score.
8. Interpreting the reader's trace precisely. Proven by the data: hops 2, 4, 5 and 6 are RFC 1918 addresses, so those provider interfaces are privately numbered; hop 3 at 137.97.29.249 is a public address on a provider interface. Suggested but not proven: that the reader's own traffic is subject to carrier-grade translation. Not shown at all: which specific device performs any translation, or where exactly it sits.
9. Hops 5 and 6 listing the same two addresses in different order, 172.16.18.33 and 172.26.14.75, is the signature of equal-cost multi-path forwarding, where consecutive probes take different parallel links. It is normal. Paris traceroute, published in 2006 by Brice Augustin and colleagues, exists specifically to keep the flow identifier constant so that this artefact disappears.
10. Definitive detection, not inference: read the WAN address from the router's own status page or with a UPnP query, then compare it with what a public reflector reports. Disagreement means an outer NAT. Both being equal and public means no CGNAT.

```
# what the internet sees
curl -s https://api.ipify.org; echo
# an alternative that uses DNS instead of HTTP
dig +short myip.opendns.com @resolver1.opendns.com
# then compare with the WAN address shown by the router itself
```

24.9.6 WORDS - remember these

1. Carrier-grade NAT — your provider sharing one public address between many customers — CGN or LSN, specified for behaviour by RFC 6888, BCP 127.
2. Double NAT — two translations between you and the internet — NAT444, three address realms with a translation at each boundary.

3. Shared address space — the block used between the carrier NAT and your router — 100.64.0.0/10 per RFC 6598, BCP 153.
4. Port block allocation — giving each customer a fixed slice of port numbers — deterministic port assignment, which reduces logging volume enormously.
5. Shared reputation — being punished for a stranger on the same address — the collapse of the one-address-one-subscriber assumption used by abuse scoring.
6. ECMP — several equally good parallel paths — equal-cost multi-path forwarding, the reason one traceroute hop can show several addresses.
7. DS-Lite — carrying IPv4 over an IPv6-only access network — Dual-Stack Lite, RFC 6333, with translation at a carrier AFTR.

24.10 Reading the reader's own first six hops as a story

24.10.1 PLAIN - in simple words

1. A traceroute is a list of the machines that handled your packet on the way out, in order.
2. It works by sending packets with a deliberately short lifespan. The first has a lifespan of one hop, so the first router kills it and sends back a complaint. The complaint reveals that router's address.
3. Then it sends one with a lifespan of two, and the second router complains. And so on.
4. So each line of a traceroute is one router saying "I am here, and your packet died on my desk".
5. The reader's first six hops tell a clear story: out of the flat, into the provider, through the provider's private core, with one public address appearing in the middle.
6. From hop 7 onwards the names change completely and the path enters Microsoft's own network, which is where GitHub lives.
7. This section reads only the first six, because those six are entirely about addressing, which is this chapter's subject.

24.10.2 PLAIN - a picture in your head

1. Imagine posting a letter with an instruction on the envelope: "after passing through one sorting office, stop and send this envelope back to me with the office's stamp on it".
2. You get back an envelope stamped by your local office. Now you know the first step of the route.
3. Post another saying "stop after two offices". Back it comes, stamped by the regional depot. Now you know the second step.
4. Repeat twenty times and you have mapped the whole postal route without ever delivering a real letter.
5. Some offices refuse to send anything back. Those steps come back blank. It does not mean the office is broken; it means it does not answer surveys.

Where this comparison breaks: postal routes are stable, so twenty separate letters follow the same path. Network paths are not. Each probe can take a different parallel link, which is exactly why hops 5 and 6 of the reader's trace show several addresses each. A traceroute is a rough sketch made from twenty different journeys, not a photograph of one.

24.10.3 PLAIN - a worked example

1. Here is the reader's own trace, hop by hop, classified.

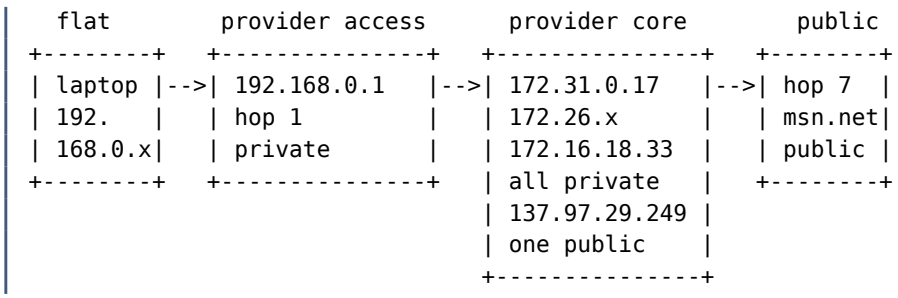
Hop	Address	Type
1	192.168.0.1	private, 192.168.0.0/16
2	172.31.0.17	private, 172.16.0.0/12
3	137.97.29.249	public
4	172.26.22.235	private, 172.16.0.0/12

Hop	Address	Type
5	172.16.18.33 and two more	private, 172.16.0.0/12
6	172.26.14.75 and one more	private, 172.16.0.0/12

2. And what each one means.

Hop	What it means
1	the reader's own router
2	the provider's first router
3	a provider public interface
4	provider core, private
5	parallel links, load balanced
6	the same parallel links again

3. Hop 5 listed 172.16.18.33, 172.26.14.75 and 172.26.22.234. Hop 6 listed 172.26.14.75 and 172.16.18.33.
4. The same addresses appear at two consecutive hops. That looks alarming and is not. It is what happens when three probes for hop 5 and two probes for hop 6 each pick a different parallel link.
5. Notice also that two different private blocks are in use: 172.16.x, 172.26.x and 172.31.x. All three are inside 172.16.0.0/12. A large provider commonly divides that /12 by region or by role.
6. The shape of the path in one picture:



24.10.4 PLAIN - what is really happening inside

1. The lifespan number on the packet is called TTL, Time To Live. Every router subtracts one before forwarding.
2. When a router subtracts one and gets zero, it must discard the packet and send back an ICMP time-exceeded message.
3. That reply comes from whichever interface address the router chooses, normally the one closest to you. That is why the addresses you see are the routers' inbound interfaces.
4. It also means a private address in a trace is a router's own interface, not your packet's source. This is the point that section 24.9 rested on.
5. Not every router replies. Some are configured not to. Some rate-limit these replies to a handful per second, which is why long traces show occasional gaps in the middle.
6. Silence at the end of a trace, which the reader saw at hops 13 to 20 as three asterisks each, is normal and proves nothing on its own.
7. The reason the reader's trace still mattered is what it showed at the start, not what it failed to show at the end: a provider with a large private core, one public interface early on, and parallel links.

24.10.5 TECHNICAL - the engineer's version

1. Traceroute was written by Van Jacobson in 1987 at Lawrence Berkeley National Laboratory. The behaviour it depends on, the ICMP time-exceeded reply, is in RFC 792, September 1981.

2. Classic Unix traceroute sends UDP datagrams to high, unlikely destination ports, incrementing the TTL. macOS `traceroute` does this by default and needs no privilege for that mode.
3. Windows `tracert` sends ICMP echo requests instead. This matters, because networks often treat ICMP and UDP differently, so the two tools can produce different paths on the same network.
4. `traceroute -I` uses ICMP on macOS and requires root. `traceroute -T` on Linux uses TCP SYN, which is often the only method that survives firewalls that discard everything else.
5. By default traceroute sends three probes per TTL value, which is why three addresses can appear on one line and why unanswered hops print three asterisks.
6. Multiple addresses on one hop indicate ECMP load balancing, typically hashed per flow on the 5-tuple. Classic traceroute varies the destination port between probes, which changes the hash, which is why the probes diverge. Paris traceroute, presented in 2006 by Brice Augustin and colleagues, holds the flow identifier constant to avoid this.
7. Reverse DNS is why hops 7 onward in the reader's trace showed names such as `ae66-0.del01-96cbe-1b.ntwk.msn.net` while hops 1 to 6 showed none. Private addresses have no public reverse record, and many carriers do not publish names for internal interfaces.
8. The naming on those later hops is a Microsoft convention, not a standard. The city codes are airport-style: `del01` Delhi, `bom01` Mumbai, `pnq20` and `pnq21` Pune. The role codes `ibr`, `rwa` and `owr` mean internal backbone, regional wide-area and outer wide-area router respectively, again by that operator's convention. Interface prefixes `ae`, `be` and `po` all mean several physical links bonded into one logical link; different vendors use different words for the same idea.
9. Useful invocations:

```
traceroute -n 20.207.73.82     # numeric only, no reverse DNS
traceroute -I -n 20.207.73.82  # ICMP probes, needs sudo on macOS
traceroute -q 1 -n 20.207.73.82  # one probe per hop, less noise
mtr -n 20.207.73.82            # continuous, shows loss per hop
```

10. The strongest reading discipline: separate what a trace proves from what it suggests. It proves which interfaces answered. It suggests, but never proves, the forward path, and it says nothing at all about the reverse path, which can be completely different.

24.10.6 WORDS - remember these

1. Traceroute — a tool that lists the routers on the way — a TTL-incrementing probe tool relying on ICMP time-exceeded replies, RFC 792.
2. TTL — the countdown that stops packets looping forever — Time To Live, an 8-bit IPv4 header field decremented at each hop.
3. ICMP time-exceeded — the complaint a router sends when a packet expires — ICMP type 11, the message that makes traceroute possible.
4. Hop — one router along the path — one decrement of TTL.
5. Asterisk in a trace — no reply came back — an unanswered probe, commonly caused by ICMP rate limiting or policy, not by a fault.
6. Reverse DNS — turning an address back into a name — a PTR record lookup in the in-addr.arpa zone.

24.11 Static and dynamic addresses, and how yours is chosen

24.11.1 PLAIN - in simple words

1. Nobody typed 192.168.0.x into the reader's laptop. It was handed over automatically.
2. The thing that hands it over is called DHCP. Your laptop shouts "does anyone have an address for me", and the router answers.
3. The router does not give the address away permanently. It lends it, for a period called a **lease**.
4. Before the lease runs out, your machine asks to keep it, and normally gets the same one back. That is why your laptop's address usually looks stable.

5. The same thing happens one level up. Your router asks your provider for a public address, and gets one on loan.
6. That loan can end. When it does, you may get a different public address. This is why home connections are said to have a **dynamic** address.
7. A **static** address is one that never changes, arranged deliberately and usually paid for.
8. If you want to run something at home that people can find by name, and your address keeps changing, you need a service that updates the name every time the address moves. That is dynamic DNS.
9. Businesses pay for static addresses because servers, mail and remote access all break when the address moves.

24.11.2 PLAIN – a picture in your head

1. Think of a library that lends reading-room desks rather than selling them.
2. You arrive, ask for a desk, and are given number 24 for four hours.
3. Nobody else gets desk 24 while your loan runs.
4. Halfway through, at two hours, you are asked whether you want to keep it. You say yes and the four hours restart. You keep the same desk.
5. If you leave without saying anything, the desk goes back into the pool once the four hours are up, and someone else may get it.
6. A subscriber who pays extra gets a reserved desk with their name on it, every day, guaranteed. That is a static address.

Where this comparison breaks: the library knows you by face. DHCP identifies you by a hardware identifier your machine sends, and modern phones and laptops deliberately randomize that identifier for privacy. So your own device may be handed a different address on purpose, on a network it has used for years, and that is a feature rather than a fault.

24.11.3 PLAIN – a worked example

1. The four steps of DHCP, in order, with the names engineers use.

Step	Who sends	Meaning
DISCOVER	the laptop	anyone out there
OFFER	the router	you can have this one
REQUEST	the laptop	yes, I take it
ACK	the router	confirmed, here are details

2. The first message goes from source 0.0.0.0 to destination 255.255.255.255, because the laptop has no address and does not know the server's address.
3. The ACK carries more than an address. It typically carries the mask, the gateway, the DNS servers and the lease time.
4. For the reader that would be: address 192.168.0.x, mask 255.255.255.0, gateway 192.168.0.1, and a DNS server.
5. Note a detail from the reader's own session: their DNS resolver was 1.1.1.1, Cloudflare's public resolver, not the router. So either the router was configured to hand out 1.1.1.1, or the laptop was set manually and ignored what DHCP offered. DHCP suggests; the operating system may override.
6. On macOS you can see the whole lease, raw:

```
ipconfig getpacket en0
# yiaddr = 192.168.0.24
# subnet_mask (ip): 255.255.255.0
# router (ip_mult): {192.168.0.1}
# lease_time (uint32): 0x15180        (86400 seconds = 24 hours)
```

7. `yiaddr` is the field name in the protocol and means "your address". It has been called that since 1993.
8. For the public side, many home connections in India and elsewhere use PPPoE: the router logs in with a username and password, and the address is negotiated as part of that session rather than by DHCP.
9. With PPPoE, the address usually changes when the session drops and reconnects, which is why rebooting a router often changes your public address, and sometimes does not.

24.11.4 PLAIN - what is really happening inside

1. A DHCP lease has three timers, not one.
2. The full lease time is how long the address is valid.
3. At half the lease time, the client tries to renew directly with the server that gave it the address. This is called T1.
4. At seven-eighths of the lease time, if renewal failed, the client broadcasts to any server that will listen. This is called T2.
5. If nothing answers by the end, the client must stop using the address. In practice most operating systems then fall back to a link-local 169.254.x.x address, which is the symptom described in section 24.4.
6. Because renewal starts at half the lease, a machine that stays switched on keeps the same address indefinitely. Addresses only really move when a machine is away long enough for the lease to expire and be reissued.
7. On the public side with PPPoE, the address is negotiated by a sub-protocol during session setup, not by DHCP at all. The provider's access concentrator picks an address from its pool and offers it.
8. Providers vary in how sticky that is. Some effectively give the same address for months. Some rotate it every 24 hours deliberately.
9. Under carrier-grade NAT the question changes shape entirely. Your router's outside address may be stable while the public address the world sees changes whenever the carrier NAT reassigns you.

24.11.5 TECHNICAL - the engineer's version

1. DHCP is RFC 2131, March 1997, with options in RFC 2132. It extends BOOTP from RFC 951, September 1985, and reuses its packet format, which is why the field is still called `yiaddr`.
2. The client sends from UDP port 68 to UDP port 67. The exchange is DHCPDISCOVER, DHCPOFFER, DHCPREQUEST, DHCPACK, commonly abbreviated DORA.
3. Renewal timers default to T1 at 0.5 of the lease and T2 at 0.875 of the lease, per RFC 2131. Both can be set explicitly with options 58 and 59.
4. Typical lease durations by deployment:

Environment	Typical lease
Home router	24 hours
Corporate wired	8 days
Public Wi-Fi, cafe	1 to 2 hours
Conference network	10 to 30 minutes

5. Client identity is normally the MAC address in the `chaddr` field, optionally overridden by option 61. Since Android 10 in 2019 and iOS 14 in 2020, and in recent macOS releases on the desktop, devices rotate a randomized MAC per network by default. Address reservations keyed on MAC therefore need the randomization turned off for that network.
6. A DHCP reservation, sometimes called static DHCP, pins one address to one client identifier. It is not the same as a static address configured on the host: the client still leases, it simply always receives the same value.
7. PPPoE is RFC 2516, February 1999. It carries PPP over Ethernet; the address is assigned during the IPCP negotiation phase of PPP, from RFC 1332. The provider side is a BRAS or BNG, a broadband remote access server or broadband network gateway.

8. Dynamic DNS updates are standardized in RFC 2136, April 1997. Consumer services predate and largely ignore it, using proprietary HTTP endpoints instead; most home routers have built-in support for two or three of them. Practical caveat: dynamic DNS is useless behind carrier-grade NAT, because the name will resolve to an address that will not accept your inbound traffic.
9. Static addresses cost money for a reason that has nothing to do with technology. They consume a scarce public address permanently, whereas dynamic pools let a provider serve more subscribers than it has addresses, because not all are online at once. That over-subscription is the whole economic point.
10. Observation commands:

```
ipconfig getpacket en0             # the full DHCP lease on macOS
ipconfig getoption en0 lease_time  # just the lease duration
sudo ipconfig set en0 DHCP         # force a fresh DHCP exchange
netstat -rn | grep default         # the gateway the lease installed
```

24.11.6 WORDS - remember these

1. DHCP — the system that hands out addresses automatically — Dynamic Host Configuration Protocol, RFC 2131, UDP ports 67 and 68.
2. Lease — a borrowed address with an expiry — a time-bounded binding, renewed at T1 and rebound at T2.
3. DORA — the four-step handshake — DISCOVER, OFFER, REQUEST, ACK.
4. Static address — one that never changes — a manually configured or permanently allocated address, usually a paid service.
5. Dynamic address — one that can change — an address from a provider pool, held only for the duration of a lease or session.
6. Reservation — always getting the same address from DHCP — a static mapping from client identifier to address inside the DHCP server.
7. PPPoE — logging in to your broadband with a username and password — PPP over Ethernet, RFC 2516, with addressing negotiated by IPCP.
8. Dynamic DNS — a name that follows your changing address — automated updating of an A record, standardized as RFC 2136, defeated by carrier-grade NAT.

24.12 IPv6 in brief

24.12.1 PLAIN - in simple words

1. IPv6 is the replacement for the addressing scheme this whole chapter has described. It uses 128 bits instead of 32.
2. Four times the bits does not mean four times the addresses. Each extra bit doubles the count. So 128 bits gives about 340 undecillion addresses, a 3 followed by 38 digits.
3. That is enough to give every grain of sand on earth its own network, with room left over. The size is deliberate and slightly absurd, so that nobody ever has to do this again.
4. Because the numbers are so long, they are written in hexadecimal, which uses the digits 0 to 9 and the letters a to f, in eight groups separated by colons.
5. There are two shortening rules, and only two, so the addresses stay readable.
6. IPv6 exists because IPv4 ran out, exactly as section 24.5 described.
7. With enough addresses for everyone, NAT is no longer needed. Every device can have its own real address again.
8. The reader's own machine reported IPv6 as `(none)`. That means their connection offered no IPv6 at all, so every packet in this chapter's story travelled over IPv4, through the provider's private core and its translation.
9. Chapter 34 covers IPv6 properly. This section is only enough to place it in the story of addressing.

24.12.2 PLAIN - a picture in your head

1. Think of telephone numbers being replaced by a scheme with room for every person, object and appliance on the planet.
2. The new numbers are longer, so nobody memorizes them. You rely on your contacts list. In networking that contacts list is DNS.
3. The old and the new cannot dial each other. A new-scheme phone cannot ring an old-scheme phone at all.
4. So during the change, every phone must be able to use both schemes. That is dual stack.
5. Nobody wants to be first, because being first costs money and buys nothing until others follow. Everyone wants to be second.

Where this comparison breaks: phone numbers could be translated between schemes by an exchange, and countries did exactly that when they lengthened numbers. IPv4 and IPv6 are genuinely different protocols with different header layouts, and translation between them is lossy and awkward. That incompatibility, not laziness, is the main reason the change has taken three decades.

24.12.3 PLAIN - a worked example

1. Full form, all 32 hex digits written out:

   ```
   2001:0db8:0000:0000:0000:ff00:0042:8329
   ```

2. Rule one: drop leading zeros inside each group. Not all zeros, only leading ones.

   ```
   2001:db8:0:0:0:ff00:42:8329
   ```

3. Rule two: replace the longest run of all-zero groups with a double colon, once and only once in an address.

   ```
   2001:db8::ff00:42:8329
   ```

4. Once only, because two double colons would be ambiguous. A reader could not tell how many zero groups belonged to each.
5. A second example, a link-local address of the kind macOS shows on every interface:

   ```
   fe80:0000:0000:0000:0204:61ff:fe9d:f156
   fe80::204:61ff:fe9d:f156
   ```

6. Note that `0204` became `204`, losing only the leading zero, and that `61ff` was untouched because its zeros are not leading.
7. Two special short ones:

Meaning	Full	Short
Loopback	0000: … :0001	::1
Unspecified	0000: … :0000	::

8. The main address types, by their leading bits:

Type	Prefix	Use
Global unicast	2000::/3	routable on the internet
Link-local	fe80::/10	this link only, always present
Unique local	fc00::/7	private, like RFC 1918
Multicast	ff00::/8	groups

Type	Prefix	Use
Documentation	2001:db8::/32	books and manuals

9. Every IPv6 interface has a link-local address whether or not it has anything else. That is required, not optional, and it is what neighbour discovery runs over.

24.12.4 PLAIN - what is really happening inside

1. IPv6 does not use DHCP by default. A machine can configure itself.
2. It works like this. The machine sends a router solicitation on the link. A router replies with a router advertisement carrying a prefix, usually 64 bits long.
3. The machine takes that 64-bit prefix and invents its own 64-bit second half. Joining them gives a complete address.
4. That is called SLAAC, stateless address autoconfiguration. Stateless because no server keeps a record of who has what.
5. Originally the second half was derived from the hardware address, which meant the same machine had a recognizable address on every network it visited. That is a privacy disaster: you could be tracked across cafes and cities.
6. So privacy addresses were added. The machine generates random second halves, uses one for outgoing connections, and replaces it every day or so.
7. This is why a machine with IPv6 usually shows several addresses at once on one interface: a link-local one, a stable one, and one or more temporary ones.
8. During the transition most machines run both protocols side by side. When you open a connection to a name that has both kinds of address, the machine races them and uses whichever answers first.
9. The reader's machine had no IPv6 at all, so none of this applied. Every attempt to reach 20.207.73.82 went over IPv4, and had no alternative path to fall back on.

24.12.5 TECHNICAL - the engineer's version

1. IPv6 is specified in RFC 8200, July 2017, Internet Standard STD 86, which obsoletes RFC 2460 from December 1998. The first specification was RFC 1883, December 1995.
2. The addressing architecture is RFC 4291, February 2006. The canonical text representation is RFC 5952, August 2010.
3. RFC 5952 states the rules as requirements, not preferences: leading zeros must be suppressed; `::` must shorten the longest zero run; `::` must not be used for a single zero group; on a tie the leftmost run wins; hexadecimal letters must be lowercase.
4. Address space: 2 to the power 128, exactly 340,282,366,920,938,463,463,374,607,431,768,211,456. Global unicast is currently allocated only out of 2000::/3, which is one eighth of the total, leaving the rest untouched for the future.
5. Unique local addresses are RFC 4193, October 2005. The block is fc00::/7, but in practice only fd00::/8 is used, with 40 random bits chosen by the site so that two organizations merging almost certainly do not collide. That random collision-avoidance is the direct lesson learned from RFC 1918 overlap.
6. SLAAC is RFC 4862, September 2007, using ICMPv6 Neighbor Discovery from RFC
4861. Temporary privacy addresses are RFC 8981, February 2021, which obsoletes RFC 4941 from September 2007.
7. Dual stack is RFC 4213, October 2005. Happy Eyeballs version 2, RFC 8305, December 2017, is the algorithm that races an IPv6 and an IPv4 connection attempt with a short head start for IPv6, and is why a broken IPv6 path no longer stalls a browser for thirty seconds.
8. Header differences that matter in practice: the IPv6 header is a fixed 40 bytes with no options inline and no header checksum; routers never fragment, so path MTU discovery is mandatory; the minimum link MTU is 1280 bytes.
9. Why thirty years:

Reason	Effect
No wire compatibility	v6-only cannot reach v4-only
No first-mover benefit	cost now, benefit later
NAT relieved pressure	the crisis felt survivable
Equipment and staff cost	every device and skill set
Dual stack doubles work	two of everything to operate

10. Measured adoption is published live by Google as the share of its users reaching it over IPv6. It crossed 10 percent in 2016 and has grown steadily since; because that figure moves every month, look it up rather than trusting a printed number, including this one.
11. Observation on macOS: `ifconfig en0 inet6`, `netstat -rn -f inet6`, and `ping6 -c1 ipv6.google.com`. If the last of these fails and `ifconfig` shows only an `fe80::` address, you have link-local IPv6 only and no global connectivity, which is exactly the reader's situation reported as `IPv6: (none)`.

24.12.6 WORDS - remember these

1. IPv6 — the 128-bit replacement for IPv4 — Internet Protocol version 6, RFC 8200, STD 86.
2. Hextet — one group of four hex digits — a 16-bit field of an IPv6 address, eight per address.
3. Zero compression — the double colon shortcut — `::` replacing the longest run of all-zero 16-bit fields, once per address, per RFC 5952.
4. Global unicast — an address the internet can route to — currently allocated from 2000::/3.
5. Link-local — the address every IPv6 interface always has — fe80::/10, used by neighbour discovery, never routed off the link.
6. Unique local — IPv6's version of a private address — fc00::/7, used as fd00::/8 with 40 random bits, RFC 4193.
7. SLAAC — a machine configuring its own address from a router hint — stateless address autoconfiguration, RFC 4862.
8. Privacy address — a temporary random address that changes daily — temporary address extensions, RFC 8981.
9. Dual stack — running both protocols at once — RFC 4213, with connection racing by Happy Eyeballs, RFC 8305.

24.13 Practical: finding your own addresses on macOS

24.13.1 PLAIN - in simple words

1. Four facts describe your position on the network: your own address, your mask, your gateway, and your public address.
2. Your own address is what your machine is called on the local network.
3. Your mask says how big that local network is.
4. Your gateway is the router you hand everything else to. For the reader it is 192.168.0.1.
5. Your public address is what the rest of the internet sees. It is usually different from your own address, because of NAT.
6. Getting all four takes about thirty seconds and four commands.
7. Comparing two of them tells you instantly whether your provider is running carrier-grade NAT.

24.13.2 PLAIN - a picture in your head

1. Think of checking your own postal details before complaining that a parcel never arrived.
2. Your flat number is your local address. The building's street number is your public address. The porter is the gateway. The postcode area is the mask.
3. If someone tells you the parcel was delivered to a street number you have never heard of, then something in the middle is redirecting your post, and you need to know that before you argue with anyone.

Where this comparison breaks: you can walk outside and read your building's street number. You cannot see your public address from inside your own machine at all. You must ask a server on the internet to tell you what it sees. That is not a convenience, it is a fundamental consequence of NAT.

24.13.3 PLAIN - a worked example

1. The four commands, in order.

```
# 1. my own address on Wi-Fi
ipconfig getifaddr en0
# 192.168.0.24

# 2. my mask, and the address again, with more detail
ifconfig en0 | grep 'inet '
# inet 192.168.0.24 netmask 0xffffff00 broadcast 192.168.0.255

# 3. my gateway
netstat -rn -f inet | grep default
# default   192.168.0.1   UGScg   en0

# 4. what the internet sees
curl -s https://api.ipify.org; echo
```

2. Note that macOS prints the mask in hexadecimal. `0xffffff00` is 255.255.255.0, which is a /24, which matches section 24.2 exactly.
3. Converting is easy: `ff` is 255, `00` is 0. So `0xffffff00` reads as 255, 255, 255, 0.
4. One command gives three of the four at once:

```
networksetup -getinfo "Wi-Fi"
# IP address: 192.168.0.24
# Subnet mask: 255.255.255.0
# Router: 192.168.0.1
```

5. Now the carrier-grade NAT test. Take the public address from step 4. Then open your router's status page and read the address on its outside, or WAN, interface.
6. Decide as follows:

```
Is the router's WAN address in 100.64.0.0/10,
10.0.0.0/8, 172.16.0.0/12 or 192.168.0.0/16 ?
   |
   +-- yes -> you are behind carrier-grade NAT.
   |
   +-- no --> is it the same as what curl reported ?
                |
                +-- yes -> a normal public address, no CGNAT
                |
                +-- no --> something translates in between:
                           CGNAT, or a proxy, or a VPN
```

7. Do the VPN check first, because it produces the same symptom. The reader's machine showed several `utun` interfaces; if one of them is active and carrying the default route, the public address you see is the VPN's, not your provider's.

```
ifconfig | grep -A2 '^utun'      # are any tunnels up
netstat -rn -f inet | grep default  # which interface is default
```

24.13.4 PLAIN - what is really happening inside

1. `ipconfig getifaddr` reads the address the operating system has stored for that interface. No packets are sent. It is instant and works offline.

2. `netstat -rn` prints the routing table straight from the kernel. The line marked `default` is the route used for anything not matching a more specific entry.
3. `curl https://api.ipify.org` is completely different in kind. It makes a real connection to a real server on the internet, and that server reports the source address it saw.
4. So that command measures what the world sees after every translation on the path, including your router, any carrier NAT, and any VPN.
5. The DNS-based alternative asks a resolver the same question over DNS rather than HTTP. It is worth knowing because it works when HTTP is blocked but DNS is not, and because it takes a different path.
6. If those two methods disagree, that itself is information: something is treating your web traffic and your DNS traffic differently.
7. None of these commands can see the carrier NAT box itself. You infer it from the difference between two numbers, which is the only tool you have from inside.

24.13.5 TECHNICAL - the engineer's version

1. Command reference, macOS, with what each actually queries:

Command	Source of truth
ipconfig getifaddr en0	configd stored state
ifconfig en0	kernel interface list
netstat -rn -f inet	kernel routing table
route -n get default	kernel route lookup
ipconfig getpacket en0	the stored DHCP lease
networksetup -getinfo	the system preference
curl to a reflector	a live remote observation

2. The interface name is not always `en0`. On many Macs `en0` is Wi-Fi, but on models with Ethernet it may be wired, with Wi-Fi on `en1`. Resolve it properly rather than guessing:

```
networksetup -listallhardwareports
route -n get default | awk '/interface/{print $2}'
```

3. Reflector services worth knowing, all returning plain text: `api.ipify.org`, `ifconfig.me`, `icanhazip.com`. Prefer HTTPS. The DNS method, `dig +short myip.opendns.com @resolver1.opendns.com`, avoids HTTP entirely.
4. To force IPv4 or IPv6 explicitly, use `curl -4` and `curl -6`. On a dual stack host these can legitimately return different addresses, and a machine reporting `IPv6: (none)` as the reader's did will simply fail on `curl -6`.
5. For a fuller picture, `scutil --nwi` prints macOS network information including which interfaces are considered usable for IPv4 and IPv6, and `arp -an` shows the neighbours your machine has actually spoken to on the local segment, which will include 192.168.0.1.
6. If the WAN address is in 100.64.0.0/10, the diagnosis is certain and needs no further evidence: RFC 6598 space exists only for the link between a carrier NAT and customer equipment.
7. If the WAN address is RFC 1918 space, as the reader's hop 2 at 172.31.0.17 suggests may be the case here, the diagnosis is very likely but not certain, because a modem in bridge or router mode can also produce that pattern locally.
8. Practical consequence to check next: if you are behind carrier-grade NAT, stop trying to fix inbound problems with port forwarding, and ask the provider for a public address, or use an outbound-initiated tunnel service instead.

24.13.6 WORDS - remember these

1. Gateway — the router you hand outbound traffic to — the next hop of the default route, 0.0.0.0/0.
2. Default route — where anything unmatched goes — the least specific routing table entry, always losing to any longer prefix match.
3. WAN address — the address on your router's outside — the address the provider assigned to the customer equipment, by DHCP or PPPoE.
4. Reflector — a server that tells you how you look from outside — an HTTP or DNS service echoing the observed source address.
5. Hexadecimal netmask — the `0xffffff00` form macOS prints — the same 32-bit mask written in base 16, two digits per octet.
6. scutil — the macOS tool for network state — the System Configuration framework utility, `scutil --nwi` for reachability information.

24.98 Common wrong ideas

1. Wrong: an IP address identifies a person. Right: it identifies a network interface at a moment in time. Under NAT it identifies a whole household; under carrier-grade NAT it can identify hundreds of households at once.
2. Wrong: an IP address identifies a device. Right: it identifies one interface. One machine can hold many addresses at once across Wi-Fi, Ethernet, loopback and every VPN tunnel, and the reader's macOS session showed several `utun` interfaces doing exactly that.
3. Wrong: private addresses are more secure. Right: they are not routable, which is different from being protected. Anything on your own network reaches them freely, and most real attacks arrive through connections you started, which NAT permits by design.
4. Wrong: NAT is a firewall. Right: NAT drops unsolicited inbound traffic as a side effect of not knowing where to send it. A firewall makes deliberate policy decisions, inspects traffic and can be audited. Relying on NAT for security means relying on an accident.
5. Wrong: changing your IP address hides you. Right: it changes one weak identifier. Cookies, browser fingerprints, logins and DNS queries all survive the change, and under carrier-grade NAT your address was never uniquely yours to begin with.
6. Wrong: 172.16.0.0/12 means everything starting 172.16. Right: it covers 172.16.0.0 through 172.31.255.255, which is why the reader's hops at 172.31.0.17, 172.26.22.235 and 172.16.18.33 are all private.
7. Wrong: private addresses in a traceroute prove you are behind carrier-grade NAT. Right: they prove those router interfaces are privately numbered. The proof of carrier-grade NAT is comparing your router's WAN address with what a public reflector reports.
8. Wrong: my router's address 192.168.0.1 clashes with my neighbour's. Right: it cannot, because private addresses are never carried on the public internet. The two networks never meet.
9. Wrong: a hop showing three addresses is an error or a loop. Right: it is equal-cost multi-path load balancing across parallel links, which is exactly what the reader's hops 5 and 6 show.
10. Wrong: IPv6 is just IPv4 with longer addresses, so switching is easy. Right: it is a separate protocol with a different header, and an IPv6-only host cannot reach an IPv4-only host at all. That incompatibility is why the transition has taken three decades.

24.99 Chapter summary in 20 lines

1. An IPv4 address is 32 bits labelling one network interface, not a device and not a person.
2. Written as four decimal octets, giving 2 to the power 32, exactly 4,294,967,296 possible addresses.
3. 192.168.0.1 is 11000000 10101000 00000000 00000001 in binary, and 3232235521 as a single integer.
4. Every address splits into a network part and a host part; the subnet mask says where, and CIDR notation writes the mask as a count of bits.

5. For 192.168.0.0/24 the network is 192.168.0.0, the broadcast 192.168.0.255, the usable range 192.168.0.1 to 192.168.0.254, and the host count 254.
6. RFC 1918, February 1996, BCP 5, reserves 10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16 for private use.
7. 172.16.0.0/12 runs from 172.16.0.0 to 172.31.255.255, so the reader's hops at 172.31.0.17, 172.26.x and 172.16.18.33 are all private addresses.
8. Private addresses never travel on the public internet, which is why every home can use 192.168.0.1 with no conflict.
9. Other reserved blocks include 127.0.0.0/8 loopback, 169.254.0.0/16 link-local, 100.64.0.0/10 shared space for carrier NAT, and 224.0.0.0/4 multicast.
10. The IANA free pool was emptied on 31 January 2011 and declared depleted on 3 February 2011; APNIC, serving India, hit its final block on 15 April 2011.
11. Classful addressing wasted enormous space by offering only three block sizes; CIDR replaced it in 1993 and is now RFC 4632.
12. NAT rewrites the source address on the way out and restores it on the way back, using a table keyed on the 5-tuple.
13. What home routers actually run is PAT, also called NAPT or NAT overload, which rewrites the port as well as the address.
14. A port is a 16-bit number identifying one conversation, ranged 0 to 1023 well-known, 1024 to 49151 registered, 49152 to 65535 ephemeral per RFC 6335.
15. Ports are what let one public address carry tens of thousands of simultaneous conversations; without them NAT would be pointless.
16. NAT breaks inbound connections, needs port forwarding for servers, and forces peer-to-peer software to use STUN, TURN and ICE with hole punching.
17. NAT was proposed in RFC 1631 in 1994 as a short-term measure; it is honestly a workaround, not a design, and it violates the end-to-end principle.
18. Carrier-grade NAT applies the same trick at the provider, costing you inbound connections, a shared reputation and much harder diagnosis.
19. The reader's first six hops read private, private, public, private, private, private, which proves the provider numbers its core from RFC 1918 space and strongly suggests, without proving, carrier-grade translation.
20. IPv6 with 128 bits removes the shortage entirely, but the reader's machine reported IPv6 as none, so every packet in this story travelled the crowded IPv4 path; Chapter 34 takes IPv6 further.

25

CHAPTER

KB

DNS - From a Name to an Address

25.0 What this chapter gives you

1. You will be able to explain why the internet needs a naming system at all, and what people did in the years before it existed.
2. You will be able to draw the shape of the whole name space, and read `github.com` as a path starting from a single dot.
3. You will be able to name the four kinds of thing that take part in a lookup, and say exactly which one does the running around.
4. You will be able to walk through a real lookup of `github.com` from the reader's own machine, step by step, with realistic times for each step.
5. You will be able to explain the thirteen root servers honestly, including why there are only thirteen names and why there are over 1,900 machines.
6. You will be able to read every common record type and know when each is used, including the rule that catches people out at the top of a domain.
7. You will be able to explain caching and TTL well enough to plan a migration, and to say why "waiting for DNS to propagate" is the wrong picture.
8. You will be able to explain why the same name gives different answers in different countries, and separate the DNS reasons from the routing reasons.
9. You will be able to say exactly why the reader's `github.com` resolved to `20.207.73.82`, and prove the ownership of that address yourself.
10. You will be able to diagnose a DNS fault with `dig`, and tell in one command whether a problem is DNS at all or something else entirely.

25.1 The problem DNS solves

25.1.1 PLAIN - in simple words

1. Computers on a network find each other by number, not by name.
2. The number for the reader's copy of `github.com` was `20.207.73.82`.
3. People are bad at remembering numbers and good at remembering names.
4. So we need something that turns a name into a number, on demand, in a few thousandths of a second, billions of times a day, all over the world.
5. That something is the **Domain Name System**, almost always called DNS.
6. DNS is a lookup service. You give it a name. It gives you back a number.
7. It is not a directory you browse. There is no list of every name. You can only ask about a name you already know.
8. DNS does one small job, and does it for the entire internet. Almost nothing else on the internet works if DNS stops working.

9. That is why a DNS fault feels like "the internet is down" even when every cable and every router is perfectly fine.

25.1.2 PLAIN - a picture in your head

1. Think of a very large office building with no signs on any door.
2. Every room has a number. Room 20207 on floor 73. Nothing else.
3. You want to visit a person, not a number. You know the person's name.
4. At the entrance there is a desk. You say a name. The desk says a room number. You walk to that room.
5. The desk does not walk you there. It only tells you where to go. Walking is still your job.
6. Now imagine the building is the size of a planet, with billions of rooms, and rooms move house every few minutes.
7. One desk could never hold all of that. So the building has one small desk per department, and the front desk knows which department desk to ask.
8. That chain of desks, each knowing only its own part, is the shape of DNS.

Where this comparison breaks: a receptionist knows the answer or does not. A DNS server usually knows only who to ask next, and that is the whole trick. The front desk in DNS almost never holds the final answer. Also, in a real building you ask one desk once. In DNS the answer is remembered by everyone along the way for a set number of seconds, which changes the behaviour completely.

25.1.3 PLAIN - a worked example

1. Before DNS existed, there was one file. It was called `HOSTS.TXT`.
2. It was a plain text file. Each line held one address and one name.
3. Here is roughly what a line looked like in the ARPANET era.

```
HOST : 10.0.0.51 : SRI-NIC,NIC : DEC-2060 : TOPS20 : TCP/TELNET
HOST : 10.1.0.11 : UCLA-CCN : IBM-3033 : OS/MVS : TCP
HOST : 26.0.0.73 : SRI-KL,SRI : FOONLY-F2 : TENEX : TCP/FTP
```

4. One organization maintained that file for everyone: the Network Information Center at Stanford Research Institute, in California.
5. If your machine changed address, you emailed or phoned the Network Information Center. A human edited the file.
6. Every other machine on the network downloaded the whole file, by FTP, every so often, and used its own copy.
7. Work out the cost. If there are 300 machines and each downloads the file twice a week, that is 600 transfers a week from one server.
8. Now double the number of machines. The file gets twice as big **and** twice as many machines download it. The load grows with the square.
9. Worse: your copy is stale from the moment you download it. A machine that moved on Tuesday is unreachable for you until you download again.
10. And there could be only one machine called `SRI-KL` in the world, because the file had one flat list of names with no structure.

25.1.4 PLAIN - what is really happening inside

1. The `HOSTS.TXT` design failed for four separate reasons, and it is worth keeping them apart because DNS fixes each one differently.
2. **Load.** One server served every machine. DNS fixes this by spreading the data across millions of servers, each holding a small piece.
3. **Freshness.** A downloaded copy is a snapshot. DNS fixes this by asking at the moment you need the answer, and by attaching an expiry time.

4. **Naming clashes.** One flat list means one name each. DNS fixes this by giving names a tree shape, so `www` can exist under a million parents.
5. **Human bottleneck.** One team edited every change. DNS fixes this by handing each organization control of its own branch. This is called **delegation** and it is the single most important idea in the system.
6. Delegation means the top of the tree does not know your addresses. It only knows which servers to ask about your part of the tree.
7. That keeps the top of the tree small, stable, and almost never changing, while the leaves change every minute.

25.1.5 TECHNICAL - the engineer's version

1. `HOSTS.TXT` was maintained by the Network Information Center at Stanford Research Institute, known as SRI-NIC. Elizabeth Feinler ran that group and built and maintained the first ARPANET host directory.
2. The file format and the FTP distribution procedure were written up in RFC 952 and RFC 953, both published in October 1985, which formalized a practice already years old.
3. Paul Mockapetris, then at the Information Sciences Institute of the University of Southern California, designed DNS in 1983 at the request of Jon Postel.
4. The original specification was published in November 1983 as two documents: RFC 882, "Domain Names - Concepts and Facilities", and RFC 883, "Domain Names - Implementation and Specification".
5. Those were obsoleted in November 1987 by RFC 1034 and RFC 1035, with the same two titles. Those two documents are still the base specification today.
6. Nearly forty years of later RFCs amend them. Nothing has replaced them.
7. The first widely used implementation was BIND, the Berkeley Internet Name Domain, written at the University of California, Berkeley in 1984 and later maintained by the Internet Systems Consortium.

Item	Date	Document
HOSTS.TXT format	Oct 1985	RFC 952
DNS first spec	Nov 1983	RFC 882, RFC 883
DNS current base	Nov 1987	RFC 1034, RFC 1035
Negative caching	Mar 1998	RFC 2308
EDNS0 (current)	Apr 2013	RFC 6891

8. The honest version: RFC 1034 and RFC 1035 are not a clean read. Real behaviour today is spread over roughly one hundred RFCs, and the community maintains an informal list called "DNS Terminology", RFC 9499, published in March 2024, purely to keep the vocabulary straight.

25.1.6 WORDS - remember these

1. DNS — the thing that turns names into numbers — Domain Name System, the distributed hierarchical database defined by RFC 1034 and RFC 1035.
2. Resolution — looking a name up — the process of mapping a domain name to resource records of a requested type and class.
3. Delegation — handing control of a branch to someone else — NS records in a parent zone pointing at the authoritative servers of a child zone.
4. HOSTS.TXT — the single hand-edited file everyone copied before DNS — the flat ARPANET host table distributed by FTP from SRI-NIC.
5. Zone — one chunk of the tree that one operator controls — an administrative unit of the name space, bounded above and below by delegation points.

25.2 The shape of the name space

25.2.1 PLAIN - in simple words

1. Domain names look flat when you type them. They are not.
2. `github.com` is a path, and you read it right to left.
3. At the far right, invisible, is the **root**. It is written as a single dot.
4. Left of the root comes `com`. That is a **top level domain**.
5. Left of that comes `github`. That is a **second level domain**.
6. If there were more parts, like `api.github.com`, each new part to the left is one step further down the tree.
7. Each part between the dots is called a **label**.
8. The full name, including the invisible dot at the end, is the **fully qualified domain name**, usually shortened to FQDN.
9. Written properly it is `github.com.` with a trailing dot. Nobody types that dot. Browsers and programs add it silently.
10. The tree grows downwards from the root, which is why people call it an inverted tree. The root is at the top of the picture and the end of the name.

25.2.2 PLAIN - a picture in your head

1. Think of a postal address, written the way most countries write it.
2. `14 Baker Lane, Kothrud, Pune, Maharashtra, India`.
3. You read it inwards from the left, but the sorting machine reads it outwards from the right. Country first, then state, then city, then street.
4. Nobody sorting post in Delhi needs to know every street in Pune. They need to know only "send this bag to Pune".
5. The Pune office needs to know only "send this to Kothrud".
6. Domain names are the same address, written in the machine's order. The most general part is on the right, the most specific on the left.
7. `com` is the country. `github` is the city. `api` is the street.

Where this comparison breaks: a postal hierarchy is geographic and mostly fixed. The DNS tree is administrative, not geographic, and a branch can be handed to anyone anywhere. A `.in` name can be served from Texas, and a `.com` name can be served entirely from Mumbai. Also, post offices always exist at every level, whereas in DNS a level can be a pure label with no server at all.

25.2.3 PLAIN - a worked example

1. Take the reader's own name, `github.com`, and split it into a path.

```
Written:            github.com
Fully qualified:    github.com.
Read right to left:

  .            <- the root, one empty label
  com.         <- top level domain
  github.com.  <- second level domain, delegated to GitHub

If we add a subdomain:

  api.github.com.     <- third level, GitHub's own choice
  raw.github.com.
  codeload.github.com.
```

2. Note what is *not* in that picture. `www` is not special. It is just a label somebody chose, the same as `api` or `raw`.

3. `www.github.com` is not "the website version" of `github.com`. It is a different name that happens to point at the same place.
4. We can check that with one command, and here is the real answer.

```
$ dig +noall +answer www.github.com A
www.github.com.   3257  IN  CNAME  github.com.
github.com.       60    IN  A      140.82.114.4
```

5. `www.github.com` is an alias pointing at `github.com`, which then has the address. Two names, one destination.

25.2.4 PLAIN - what is really happening inside

1. Each label is stored separately, not as one long string.
2. In a DNS message on the wire, `github.com` is encoded as a length byte, then the letters, then a length byte, then the letters, then a zero.

```
06 67 69 74 68 75 62  03 63 6f 6d  00
 6  g  i  t  h  u  b   3  c  o  m  end
```

3. The final `00` is the root label. It has length zero. That is the trailing dot, and it is genuinely present in every message on the wire.
4. So the trailing dot is not a typing convention. It is a real byte.
5. A **domain** is a name and everything under it. A **zone** is the part of a domain that one server actually holds.
6. Those two are different, and mixing them up causes real confusion. `com` is a domain containing millions of names. The `com` zone contains almost none of their addresses, only pointers to who does.
7. The point where a parent stops and a child starts is called a **zone cut**. At every zone cut there is a delegation.

25.2.5 TECHNICAL - the engineer's version

1. RFC 1035 section 2.3.4 sets the hard limits, and they have never changed.

Limit	Value	Note
One label	63 octets	Length byte uses 6 bits
Full name on wire	255 octets	Includes length bytes
Labels per name	127 max	A consequence of the above
TTL field	32 bits	Signed use, max 2147483647

2. Comparison of labels is case insensitive but case is preserved on the wire. RFC 4343, published January 2006, spells this out precisely.
3. Many resolvers exploit that by randomizing the case of the query name and checking that the case comes back unchanged. This is called 0x20 encoding. It is a convention and a defence, not a standard.
4. Top level domains come in categories: generic (gTLD) such as `com`, country code (ccTLD) such as `in`, sponsored such as `edu`, and one infrastructure TLD, `arpa`, used for reverse lookups.
5. The authoritative list is the IANA root zone database. As of version 2026081200, last updated 12 August 2026, it lists roughly 1,500 top level domains, including internationalized names that begin with `xn--`.
6. Certain names are reserved by standard and must never be delegated. RFC 2606, June 1999, reserves `example.com`, `example.net`, `example.org` and the TLDs `test`, `invalid`, `example` and `localhost`.
7. RFC 6761, February 2013, defines special use domain names that resolvers are expected to handle locally rather than by querying the tree.

8. The honest version: `.local` is not part of the global DNS at all. It is handled by Multicast DNS, RFC 6762, February 2013, which is a different protocol on UDP port 5353 that happens to reuse the DNS message format.

25.2.6 WORDS - remember these

1. Label — one piece between the dots — an octet string of 1 to 63 octets, compared case insensitively.
2. FQDN — the complete name including the final dot — a domain name terminated at the root, unambiguous with no search list applied.
3. TLD — the last part, like `com` — a zone delegated directly from the root zone by IANA.
4. Zone — the part of the tree one operator serves — the set of records under a name, stopping at each delegation below it.
5. Zone cut — the boundary between parent and child — the point where NS records in the parent delegate authority downward.

25.3 The four kinds of participant

25.3.1 PLAIN - in simple words

1. Four different things take part in a lookup. They are easy to confuse, and almost every DNS misunderstanding comes from confusing two of them.
2. The **stub resolver** lives inside your own computer. It is small. It knows almost nothing. It asks one question and waits for one answer.
3. The **recursive resolver** is a server somewhere else. It is the one that does the work. For the reader that server is at `1.1.1.1`.
4. The **root servers** sit at the very top. They know one thing: which servers handle each top level domain.
5. The **authoritative servers** hold real answers for one zone each. The servers for `com` are authoritative for `com`. GitHub's servers are authoritative for `github.com`.
6. Say this out loud, because it is the sentence people get wrong: your machine does not walk the tree. The recursive resolver walks the tree.
7. Your machine asks one question, once, and waits. Everything else happens somewhere else, at somebody else's expense.

25.3.2 PLAIN - a picture in your head

1. Imagine you want a rare book and you are not going to leave your chair.
2. You phone your local librarian and say: get me this book.
3. That librarian does not have the book. But the librarian has a phone, a membership of the national library network, and patience.
4. The librarian phones the national catalogue. The catalogue says: that subject belongs to the medical library.
5. The librarian phones the medical library. That library says: that title is in the Pune branch.
6. The librarian phones the Pune branch and gets the answer.
7. Then the librarian phones you back with one sentence, and writes the answer on a card so the next person who asks gets it instantly.
8. You made one phone call. The librarian made three. That is exactly the split between a stub resolver and a recursive resolver.

Where this comparison breaks: a librarian could read the book to you. A DNS resolver never fetches the page, only the address. It also never invents an answer, and never asks a server that is not named in a referral it just received. Also the librarian is trusted by you personally, whereas your resolver is often a company you have never had any dealings with.

25.3.3 PLAIN - a worked example

1. Here is who holds what, for the reader's actual lookup.

Who	Knows	Example
Stub on the Mac	Nothing, plus cache	asks 1.1.1.1
Recursive 1.1.1.1	Who to ask, plus cache	walks the tree
Root server	Where `com` lives	13 names, A to M
`com` servers	Where `github.com` lives	a to m gtld-servers
GitHub's servers	The actual address	dns1.p08.nsone.net

2. Notice that only the bottom row knows the answer.
3. Notice also that the top three rows never change from one lookup to the next. Only the bottom row is specific to `github.com`.
4. That is why a resolver that has been running for an hour answers most queries without leaving the building. It already knows the top three rows.

25.3.4 PLAIN - what is really happening inside

1. The stub resolver on macOS is not a separate program you can see easily. It is library code plus a system service called `mDNSResponder`.
2. When an application calls `getaddrinfo`, that call goes into the stub.
3. The stub checks a small local cache. If the answer is there and not expired, it returns immediately, with no network traffic at all.
4. If not, the stub builds one DNS query message and sends it to the configured resolver, `1.1.1.1`, on UDP port 53.
5. It sets a flag in that message called **Recursion Desired**. That flag means "please do all the work and come back with the final answer".
6. The recursive resolver honours that flag. Root, TLD and authoritative servers do not. They answer only from their own zone, or hand back a referral naming the next servers down.
7. So the difference between a recursive resolver and an authoritative server is not the software. It is the role and the configuration.
8. Many programs can do both. Running both roles on the same address is a known bad idea, because it mixes cached data with authoritative data.

25.3.5 TECHNICAL - the engineer's version

1. RFC 8499, obsoleted by RFC 9499 in March 2024, gives the precise terms: stub resolver, recursive resolver, iterative resolver, full resolver, authoritative server and forwarder.
2. A **forwarder** is a fifth role that people forget. A home router usually runs a small forwarding resolver: it accepts your query, does no tree walking, and passes the query to an upstream recursive resolver.
3. That is what the reader's router at `192.168.0.1` would normally do. The reader has bypassed it by configuring `1.1.1.1` directly.
4. The **RD** bit (Recursion Desired) is bit 8 of the DNS header flags. The **RA** bit (Recursion Available) is bit 15, set by a server that is willing to recurse for you.
5. An authoritative server sets **AA** (Authoritative Answer). A cached answer from a recursive resolver does not have AA set. This is how you tell them apart in a packet capture.
6. Query the flags directly with `dig`. In the reader's own output for `github.com` we see `flags: qr rd ra`, meaning it is a response, recursion was desired, recursion was available, and crucially there is no `aa`.
7. QNAME minimization, RFC 9156, November 2021, changed what a recursive resolver sends upward. Instead of sending the full name to the root, a modern resolver sends only `com.` to the root. Cloudflare and Google both do this by default now. This is a privacy improvement, deployed since 2019.

25.3.6 WORDS - remember these

1. Stub resolver — the tiny piece in your own machine — the client-side code that sends one recursive query and does not follow referrals.
2. Recursive resolver — the server that does the running around — a full service resolver that follows referrals and caches results.
3. Authoritative server — a server that genuinely holds a zone — one that answers from zone data and sets the AA bit.
4. Referral — "not me, ask them" — a response with no answer records and NS records for a child zone in the authority section.
5. Forwarder — a middleman that just passes queries along — a resolver configured to send all queries to another recursive resolver.
6. RD bit — "please do the work for me" — the Recursion Desired header flag, honoured only by recursive resolvers.

25.4 One full lookup, step by step

25.4.1 PLAIN - in simple words

1. We will now follow one real lookup all the way. The name is `github.com`. The machine is the reader's Mac, in India, on home broadband.
2. The resolver is `1.1.1.1`. The answer that came back was `20.207.73.82`.
3. Step 1. A program asks the operating system for the address of `github.com`.
4. Step 2. The stub resolver looks in its own small cache. Nothing there.
5. Step 3. The stub sends one question to `1.1.1.1` and waits.
6. Step 4. `1.1.1.1` looks in its own much bigger cache. Suppose it is cold and knows nothing, so it starts at the top.
7. Step 5. It asks a root server: who handles `com`?
8. Step 6. The root server does not give an address. It gives a list of the `com` servers. That is a **referral**.
9. Step 7. It asks a `com` server about `github.com`.
10. Step 8. The `com` server also gives a referral: GitHub's own name servers.
11. Step 9. It asks one of GitHub's name servers about `github.com`.
12. Step 10. That server holds the zone, so it gives a real answer.
13. Step 11. `1.1.1.1` stores the answer in its cache with a countdown.
14. Step 12. `1.1.1.1` sends the answer back to the Mac.
15. Step 13. The stub stores it too, and hands the address to the program.

25.4.2 PLAIN - a picture in your head

1. Think of finding a person in a very large company on your first day.
2. You ask reception. Reception says: that is the engineering building, ask their front desk.
3. You go to engineering reception. They say: that is the platform team, ask their team lead.
4. You go to the team lead. The team lead says: desk 42, second floor.
5. Three questions, three answers, and only the last one was useful in itself.
6. The first two were not failures. They were progress. Each one narrowed the search by a huge factor.
7. Reception did not know about desk 42, and never will, and does not need to. That is the whole design.

Where this comparison breaks: you walked between buildings. In DNS the resolver sends fresh packets to fresh servers each time, and each hop is a completely independent question with its own timeout and its own retry. Also, reception in a real company will guess if unsure. A DNS server never guesses. If it does not have a delegation, it says so, and the resolver stops.

25.4.3 PLAIN - a worked example

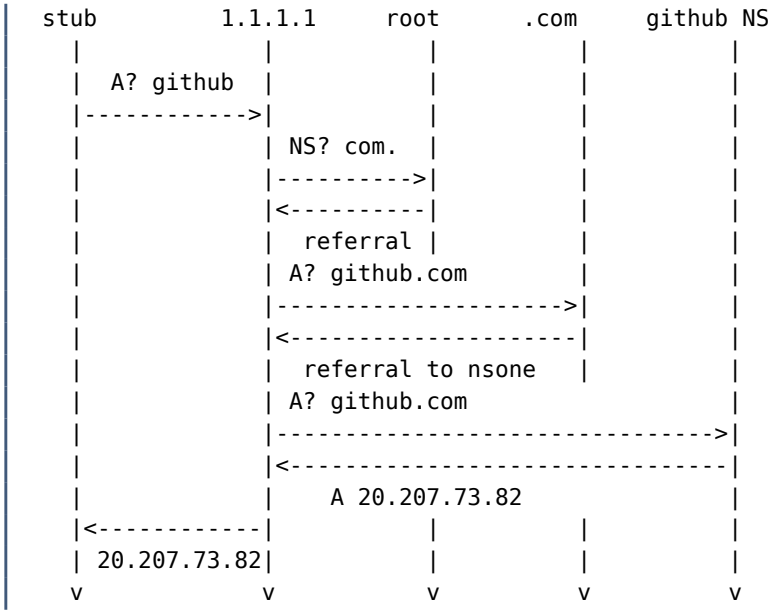
1. Here is the same lookup as a table, with the question asked at each step, what came back, and a realistic time on Indian home broadband.

Step	Question to	Answer type	Time
1	stub cache	miss	0.1 ms
2	1.1.1.1	(working)	8 ms rtt
3	root server	referral to com	18 ms
4	com server	referral to nsone	25 ms
5	nsone server	20.207.73.82	30 ms
6	back to Mac	final answer	8 ms

2. Cold total, nothing cached anywhere: roughly 90 milliseconds.
3. Warm total, when `1.1.1.1` already has the answer: roughly 8 milliseconds, because only step 2 and step 6 happen.
4. Warm total when the Mac itself has it cached: under 1 millisecond, with no packets sent at all.
5. In real life the cold case is rare. The root and `com` referrals live in a busy resolver's cache for hours, so usually only the last hop is needed.

25.4.4 PLAIN - what is really happening inside

1. Here is the same journey drawn out. Read downwards. Each arrow is one UDP packet in one direction.



2. Count the packets. Eight in total for a completely cold lookup.
3. Only two of them touch the reader's own broadband line. The other six are sent by Cloudflare, from Cloudflare's own network, at Cloudflare's cost.
4. That is the point of the design. The expensive work happens once, near the resolver, and is then reused for every other customer of that resolver.
5. Each referral arrives with **glue**: the addresses of the servers named in the referral, in the additional section. Without glue the resolver would have to look up the name servers' own names, which could loop forever.
6. If a server does not reply within about 800 milliseconds, the resolver tries another server from the same list. There are always several.

25.4.5 TECHNICAL - the engineer's version

1. You can watch the whole chain with `dig +trace`, which asks each level in turn from your own machine instead of asking a recursive resolver.
2. The output starts with the root zone NS set, which `dig` gets from its built-in hints file, then follows each referral.

```
$ dig +trace github.com

.        87203  IN  NS  a.root-servers.net.
.        87203  IN  NS  b.root-servers.net.
.        87203  IN  NS  c.root-servers.net.
...
;; Received 525 bytes from 198.41.0.4#53(a.root-servers.net)

com.     172800 IN  NS  a.gtld-servers.net.
com.     172800 IN  NS  b.gtld-servers.net.
...
;; Received 1174 bytes from 192.5.6.30#53(a.gtld-servers.net)

github.com.  172800 IN NS dns1.p08.nsone.net.
github.com.  172800 IN NS ns-421.awsdns-52.com.
...
;; Received 322 bytes from 192.31.80.30#53(d.gtld-servers.net)

github.com.  60  IN  A  20.207.73.82
;; Received 55 bytes from 198.51.44.8#53(dns1.p08.nsone.net)
```

3. The exact addresses in the final line depend on where you are. In India the reader got `20.207.73.82`. From a machine in the United States the same command returns addresses in `140.82.112.0/20`, which is GitHub's own space. Section 25.10 explains that difference in full.
4. The NS set for `github.com` is genuinely split across two providers. This is real output, taken with `dig`, and it is worth studying.

```
$ dig +noall +answer github.com NS
github.com.  1348  IN  NS  dns1.p08.nsone.net.
github.com.  1348  IN  NS  dns2.p08.nsone.net.
github.com.  1348  IN  NS  dns3.p08.nsone.net.
github.com.  1348  IN  NS  dns4.p08.nsone.net.
github.com.  1348  IN  NS  ns-421.awsdns-52.com.
github.com.  1348  IN  NS  ns-520.awsdns-01.net.
github.com.  1348  IN  NS  ns-1283.awsdns-32.org.
github.com.  1348  IN  NS  ns-1707.awsdns-21.co.uk.
```

5. Four servers at NS1, a managed DNS provider, and four at Amazon Route 53. Two independent providers, so an outage at either one does not take `github.com` off the internet.
6. Notice the Amazon servers are spread across `com`, `net`, `org` and `co.uk`. That is deliberate: if one of those TLDs has trouble, the others still resolve. This is a convention among large operators, not a standard.
7. Timing figures worth knowing, measured with `dig` and reported in the `Query time` line at the bottom of every response:

Case	Typical query time
Local stub cache hit	under 1 ms
Warm 1.1.1.1 in India	4 to 15 ms
Cold, full tree walk	60 to 150 ms
Server not responding	800 ms then retry

Case	Typical query time
Total giving up	5 s default in dig

25.4.6 WORDS - remember these

1. Referral — "ask these servers instead" — a delegation response containing NS records for a child zone and no answer records.
2. Glue record — the address of a name server, given along with its name — an A or AAAA record in the additional section, needed to break a dependency loop.
3. Iterative query — asking one level at a time — a query with the RD bit clear, answered from local data or with a referral.
4. Priming — how a resolver finds the root — a startup query for the root NS set, seeded from a built-in hints file.
5. Query time — how long the answer took — the round trip measured by dig, printed in milliseconds at the foot of the response.

25.5 The root servers

25.5.1 PLAIN - in simple words

1. At the top of the tree sit the root servers. Every lookup that is not already cached starts there.
2. There are thirteen of them, named with the letters A to M, at addresses ending in `root-servers.net`.
3. That is the sentence everyone repeats. It is true about the names, and very misleading about the machines.
4. There are not thirteen machines. There are over 1,900 physical servers spread across the world, all answering on those same thirteen addresses.
5. They are run by twelve different organizations, in different countries, under different legal systems, on purpose.
6. No single organization can switch off the root. That was designed in.
7. What the root knows is small. It knows which servers handle each top level domain, and nothing else. It has never heard of `github.com`.
8. The root zone file is only a few megabytes. You could email it.

25.5.2 PLAIN - a picture in your head

1. Think of a telephone number that reaches whichever branch of a chain is nearest to you.
2. You dial one number. Somebody answers. Which building they are sitting in depends entirely on where you are standing.
3. In Pune it rings in Pune. In Frankfurt it rings in Frankfurt. Same number, different building, and you cannot tell which.
4. That is what the thirteen root addresses are. Each one is a single number answered by hundreds of buildings.
5. The trick that makes this work is called **anycast**, and we come back to it properly in section 25.9.

Where this comparison breaks: a phone chain routes calls by a central system that decides. Anycast has no central decision at all. Each network on the path picks the shortest route it knows to that address, so the choice emerges from the routing system rather than being made by anyone. There is also no failover list: if the nearest instance disappears, routing simply converges on the next nearest, usually within seconds, with no cooperation from anybody.

25.5.3 PLAIN - a worked example

1. Here are the real thirteen, with their addresses and who runs them. This list has been almost unchanged for decades.

Name	IPv4 address	Operator
a.root-servers.net	198.41.0.4	Verisign
b.root-servers.net	170.247.170.2	USC ISI
c.root-servers.net	192.33.4.12	Cogent
d.root-servers.net	199.7.91.13	Univ of Maryland
e.root-servers.net	192.203.230.10	NASA Ames
f.root-servers.net	192.5.5.241	ISC
g.root-servers.net	192.112.36.4	US DoD NIC
h.root-servers.net	198.97.190.53	US Army Research Lab
i.root-servers.net	192.36.148.17	Netnod
j.root-servers.net	192.58.128.30	Verisign
k.root-servers.net	193.0.14.129	RIPE NCC
l.root-servers.net	199.7.83.42	ICANN
m.root-servers.net	202.12.27.33	WIDE Project

2. Verisign runs two, A and J. That is why thirteen names are run by twelve organizations.
3. One address on that list changed recently, which is unusual enough to be worth remembering: B moved from `199.9.14.201` to `170.247.170.2` on 27 November 2023.
4. Six of the thirteen are run by United States government or university bodies. Netnod is Swedish, RIPE NCC is Dutch, WIDE is Japanese. That mix is frequently argued about in internet governance, and it is fair to say the balance is historical rather than designed.

25.5.4 PLAIN - what is really happening inside

1. Each of the thirteen addresses is announced into the global routing system from many separate locations at once.
2. When your resolver sends a packet to `198.41.0.4`, the routing system delivers it to whichever announcement is closest in routing terms.
3. "Closest" here does not mean closest in kilometres. It means fewest network hops by the rules the routing protocol uses.
4. The resolver has no idea which instance answered. It sees one address.
5. You can find out, though. There is a special query, `hostname.bind` in the `CH` class, that many instances answer with their own site identifier.

```
$ dig @198.41.0.4 hostname.bind CH TXT +short
"nnn1.a.root-servers.org"
```

6. Ask that from Mumbai and from London and you will get different strings back from the same address. That is anycast made visible.
7. The root servers are astonishingly boring by design. They serve one small zone, read only, and change roughly twice a day when the root zone is republished.

25.5.5 TECHNICAL - the engineer's version

1. As of 13 August 2026 the root server system reported **2,003 operational instances** across the thirteen letters. The figure is published live by the root server operators and grows most months.
2. The distribution is very uneven. Verified counts from the same source on that date included F at 366 instances, E at 328, D at 231, I at 90, and at the other end B and G at 6 each.

3. Why thirteen, and not fourteen or a hundred? The limit came from the original 512-octet maximum for a DNS message over UDP, set in RFC 1035.
4. A priming response must carry the NS records for the root plus the A records giving their addresses. With DNS name compression, thirteen name servers plus thirteen glue addresses fit inside 512 octets. Fourteen did not.
5. That is why every root server is a single letter followed by `.root-servers.net`: the shared suffix compresses to two bytes per record.
6. The constraint is long gone. EDNS0, RFC 6891, lets a response be much larger, and the current priming response with IPv6 glue is well over 800 octets. The number thirteen survives purely because changing it has no benefit.
7. The root zone has been signed with DNSSEC since 15 July 2010. The key signing key was rolled for the first time on 11 October 2018, an operation planned for years and completed without visible breakage.
8. The root zone itself is generated by IANA, which is a function operated by Public Technical Identifiers under ICANN, and published by Verisign as the root zone maintainer.

Root fact	Value	As of
Letters	13, A to M	since 1980s
Operators	12 organizations	current
Instances	2,003	13 Aug 2026
Root zone signed	yes	15 Jul 2010
TLDs delegated	about 1,500	Aug 2026

9. The honest version: people say "the root servers are the most attacked machines on the internet". Attacks do happen, and there were notable events in October 2002 and February 2007, but the system has never gone down as a whole, and anycast means an attack is absorbed locally by the instances near the attacker rather than concentrated on one box.

25.5.6 WORDS - remember these

1. Root server — the server at the top of the name tree — an authoritative server for the root zone, reachable at one of thirteen well known addresses.
2. Anycast — one address answered from many places — the same prefix announced from multiple locations, with routing selecting the nearest.
3. Hints file — the built-in list of root addresses — the bootstrap file used for priming before any DNS query can be made.
4. Priming query — a resolver's first question at startup — a query for the root NS set, used to replace the hints with live data.
5. Root zone maintainer — the party that publishes the root file — currently Verisign, working from the zone content authorized by IANA.

25.6 Record types

25.6.1 PLAIN - in simple words

1. DNS does not only store addresses. It stores several kinds of fact about a name, and each kind has a short code called a **record type**.
2. When you ask a question you name both the name and the type. "What is the A record for `github.com`" is a different question from "what is the MX record for `github.com`", and both are normal.
3. An **A** record holds an IPv4 address. That is the common one.
4. An **AAAA** record holds an IPv6 address. Said out loud as "quad A".
5. A **CNAME** record says "this name is another name for that name".

6. An **MX** record says where to deliver email for this domain.
7. A **TXT** record holds free text, used mostly to prove ownership.
8. An **NS** record names a server that is authoritative for a zone.
9. An **SOA** record holds the administrative settings for a zone.
10. A **PTR** record maps an address back to a name.
11. An **SRV** record says which host and port run a named service.
12. A **CAA** record says which certificate authorities may issue certificates for this domain.

25.6.2 PLAIN - a picture in your head

1. Think of a person's entry in an old paper address book.
2. There is a street address. That is the A record.
3. There is a second address for their new flat abroad. That is AAAA.
4. There is a line saying "see under her maiden name". That is CNAME.
5. There is a separate line saying "send post care of the office". That is MX, because letters go somewhere different from where the person lives.
6. There is a scribbled note in the margin proving the page is genuine. TXT.
7. There is a line saying which secretary handles her diary. NS.
8. There is a stamp on the page saying who maintains this book and how often it is reprinted. SOA.
9. There is a reverse index at the back listing addresses and who lives there. PTR.

Where this comparison breaks: an address book is one document owned by one person. DNS records for one name can be split across different owners, and the reverse index at the back is maintained by an entirely different organization that may never have heard of the person. That mismatch is real and it is why reverse lookups so often disagree with forward ones.

25.6.3 PLAIN - a worked example

1. Here are real records for `github.com`, queried with `dig`. Every line below is genuine output, only trimmed for width.

```
$ dig +noall +answer github.com SOA
github.com. 3600 IN SOA dns1.p08.nsone.net.
  hostmaster.nsone.net. 1656468023 43200 7200 1209600 3600

$ dig +noall +answer github.com MX
github.com. 130 IN MX 0
  github-com.mail.protection.outlook.com.

$ dig +noall +answer github.com CAA
github.com. 3600 IN CAA 0 issue "letsencrypt.org"
github.com. 3600 IN CAA 0 issue "digicert.com"
github.com. 3600 IN CAA 0 issue "sectigo.com"
github.com. 3600 IN CAA 0 issuewild "digicert.com"

$ dig +noall +answer www.github.com A
www.github.com. 3257 IN CNAME github.com.
github.com.     60   IN A     140.82.114.4
```

2. Read the MX line. Mail for `github.com` goes to Microsoft's Outlook mail protection service, not to GitHub's own machines. The domain is GitHub's; the mail path is Microsoft's. That is a small clue about who owns what.
3. Read the CAA lines. Only four named certificate authorities may issue for `github.com`. Any other authority that follows the rules must refuse.

4. Read the SOA line. `dns1.p08.nsone.net` is the primary server, `hostmaster.nsone.net` is the contact address with the first dot standing in for an at sign, and the numbers are serial, refresh, retry, expire and minimum, in that order.

25.6.4 PLAIN - what is really happening inside

1. Records with the same name, type and class are handled together as a set, called an **RRset**. You cannot have half a set. All of it is returned or none of it.
2. A CNAME is special and slightly dangerous. It says "stop, restart the whole lookup using this other name".
3. Because a CNAME redirects everything, the standard forbids any other record existing alongside a CNAME at the same name.
4. That rule creates the problem everyone hits eventually. At the very top of a domain, called the **apex** or the **naked domain**, an SOA record and NS records must exist. They are compulsory.
5. So a CNAME can never be placed at `github.com` itself. Only at names below it, like `www.github.com`.
6. If you have ever tried to point `example.com` at a hosting provider's hostname and been told it is not allowed, that is exactly this rule.
7. The workarounds are covered in the technical block below, and it matters that some of them are not standards.

25.6.5 TECHNICAL - the engineer's version

1. The main types, with their numeric codes and defining documents.

Type	Code	Purpose
A	1	IPv4 address, RFC 1035
NS	2	Delegation, RFC 1035
CNAME	5	Alias, RFC 1035
SOA	6	Zone parameters, RFC 1035
PTR	12	Reverse mapping, RFC 1035
MX	15	Mail exchanger, RFC 1035
TXT	16	Free text, RFC 1035
AAAA	28	IPv6 address, RFC 3596
SRV	33	Service location, RFC 2782
CAA	257	CA authorization, RFC 8659

2. The CNAME restriction is RFC 1034 section 3.6.2: if a CNAME is present at a node, no other data should be present. DNSSEC records are the only exception the later standards carved out.
3. Workarounds for the apex problem, and their honest status:
    1. ALIAS or ANAME records: an **implementation detail** of particular providers such as NS1, DNSimple and Route 53. Not a standard. The provider resolves the target and serves an A record itself.
    2. CNAME flattening: Cloudflare's name for the same trick, introduced in 2014. Also not a standard.
    3. The HTTPS and SVCB record types, RFC 9460, published November 2023. These are a real standard and are supported by Chrome, Safari and Firefox, but they solve service binding rather than being a general apex CNAME.
4. `github.com` has no AAAA record. This was checked in August 2026 and the answer was NODATA, meaning the name exists but has no record of that type. The response carries an SOA record in the authority section instead.

```
$ dig +noall +answer +authority github.com AAAA
github.com. 1081 IN SOA dns1.p08.nsone.net.
  hostmaster.nsone.net. 1656468023 43200 7200 1209600 3600
```

5. That single fact explains a line in the reader's own diagnostics: their system reported `IPv6: (none)`. Even with working IPv6 at home, a connection to `github.com` over IPv6 is impossible, because no IPv6 address is published for it.
6. NODATA and NXDOMAIN are different and get confused. NXDOMAIN means the name does not exist at all. NODATA means it exists but not with that type. Both return an SOA in the authority section, which is what sets the negative cache time.
7. TXT records are the internet's junk drawer. `github.com` publishes more than twenty of them, most of them proofs of ownership for third party services.

```
"v=spf1 ip4:192.30.252.0/22 include:spf.protection.outlook.com ..."
"MS=ms44452932"
"google-site-verification=UTM-3akMgubp6tQtgEuAkYNYL..."
"apple-domain-verification=RyQhdzTl6Z6x8ZP4"
"docusign=087098e3-3d46-47b7-9b4e-8a23028154cd"
```

8. Reverse DNS lives under two special zones: `in-addr.arpa` for IPv4 and `ip6.arpa` for IPv6. The address is written backwards, label by label. `20.207.73.82` becomes `82.73.207.20.in-addr.arpa`.
9. Reverse DNS is delegated by whoever holds the address block, not by whoever holds the domain name. Those are different parties, so the two directions are free to disagree, and usually do.
10. Here are three real reverse lookups that make the point better than any explanation.

```
$ dig +short -x 140.82.112.3
lb-140-82-112-3-iad.github.com.

$ dig +short -x 8.8.8.8
dns.google.

$ dig +short -x 20.207.73.82
(no answer at all)
```

11. The first has a PTR that names GitHub and even encodes the site code `iad`, which is the airport code for Washington Dulles. The second is tidy. The third, the reader's own answer, has no PTR record whatsoever.
12. That is normal and is not a fault. Large cloud edges frequently publish no reverse records for service addresses. Mail servers are the one place where a missing or mismatched PTR causes real trouble, because receiving mail systems check it.

25.6.6 WORDS - remember these

1. RRset — all records of one type at one name — a resource record set, the atomic unit of DNS data and of DNSSEC signing.
2. Apex — the very top of a domain — the zone origin, where SOA and NS records are mandatory and a CNAME is therefore forbidden.
3. CNAME — an alias to another name — canonical name record, which restarts resolution at the target and excludes all other data at that node.
4. NODATA — the name exists but not with that type — a NOERROR response with an empty answer section and an SOA in the authority section.
5. NXDOMAIN — the name does not exist — response code 3, cached for the time given by the SOA minimum field.
6. PTR — the record that maps an address to a name — a pointer record under `in-addr.arpa` or `ip6.arpa`, controlled by the address holder.

25.7 Caching and TTL

25.7.1 PLAIN - in simple words

1. If every lookup walked the whole tree, the root servers would receive trillions of queries a day and the internet would stop.
2. They do not, because of caching. Almost every answer is reused.
3. Every DNS answer carries a number called the **TTL**, short for time to live. It is a count of seconds.
4. The TTL means: you may keep this answer and reuse it for this many seconds. After that, throw it away and ask again.
5. A resolver that just answered a question for one customer can answer the same question for a million more without doing any work.
6. This is why a busy resolver like `1.1.1.1` answers most questions in a few milliseconds. The answer was already sitting there.
7. Caching is not an optimization bolted on later. It is load bearing. DNS without caching does not work at internet scale at all.

25.7.2 PLAIN - a picture in your head

1. Think of a carton of milk with a date printed on it.
2. The date is a promise from the dairy: this is good until then.
3. You are allowed to use it any time before the date without checking.
4. After the date you must go and get fresh milk. You do not get to argue.
5. Now the important part. If the dairy changes its recipe today, every carton already in every fridge still has the old recipe until its date passes.
6. Nobody comes to your house and swaps it. There is no recall.
7. That is exactly what happens when someone changes a DNS record. Old answers sit in caches everywhere until each one expires on its own schedule.

Where this comparison breaks: milk actually goes bad. A DNS answer past its TTL may still be perfectly correct. The TTL is a policy about how long you may assume, not a statement about when the fact changes. Also, a resolver is allowed to throw the answer away early, for example when it runs short of memory. So the TTL is a maximum, not a minimum.

25.7.3 PLAIN - a worked example

1. You can watch a TTL count down with your own eyes. Ask the same question twice, a few seconds apart, through a caching resolver.

```
$ dig +noall +answer github.com A
github.com.   60   IN  A   20.207.73.82

  ... wait 10 seconds ...

$ dig +noall +answer github.com A
github.com.   50   IN  A   20.207.73.82

  ... wait 20 seconds ...

$ dig +noall +answer github.com A
github.com.   30   IN  A   20.207.73.82

  ... wait 35 seconds ...

$ dig +noall +answer github.com A
github.com.   60   IN  A   20.207.73.82
```

2. Read it carefully. The number falls by exactly one per second.
3. The resolver is not re-asking. It is subtracting the time it has held the record and reporting what is left.
4. When it reaches zero, the entry is dropped. The next query goes back to GitHub's name servers and comes back at the full 60 again.
5. `github.com` publishes a short TTL of 60 seconds. That is a deliberate choice. It lets GitHub move traffic quickly at the cost of more queries.
6. Compare that with a TTL of 86400 seconds, one day, which is common for records that never move, such as MX records for a stable mail provider.

25.7.4 PLAIN - what is really happening inside

1. There is not one cache. There are usually four or five, stacked.

```
Application (browser has its own cache)
    |
Operating system stub cache (mDNSResponder on macOS)
    |
Home router forwarder cache (often, on 192.168.0.1)
    |
Recursive resolver cache (1.1.1.1)
    |
Authoritative server (the truth)
```

2. The reader has removed one of those layers by pointing the Mac straight at `1.1.1.1` instead of at the router at `192.168.0.1`.
3. Each layer counts down independently. A record can be expired at one layer and still fresh at the layer above it.
4. Browsers make this worse. Chrome and Firefox keep their own DNS cache and have historically ignored short TTLs, holding entries for around 60 seconds regardless. That is an implementation detail and it changes between versions.
5. There is also **negative caching**: the fact that a name does not exist is itself cached, so a typo does not generate a fresh tree walk every time.
6. Negative caching uses a different number: the last field of the SOA record, called the minimum field, which was repurposed for exactly this.
7. For `github.com` that field is 3600, so a nonexistent name under `github.com` is remembered as nonexistent for up to an hour.

25.7.5 TECHNICAL - the engineer's version

1. TTL is a 32-bit field in every resource record. RFC 2181, July 1997, section 8, states it must be treated as an unsigned value between 0 and 2147483647, and that values with the top bit set must be treated as zero.
2. Negative caching is specified in RFC 2308, March 1998. The negative cache time is the **lesser** of the SOA minimum field and the SOA record's own TTL. Most implementations additionally cap it, commonly at 3 hours.
3. Real TTL values, read from live output, showing how wide the spread is:

Record	TTL seen	Meaning
github.com A	60 s	moves often
github.com NS	1348 s	of 172800 max
github.com SOA	3600 s	1 hour
github.com CAA	3600 s	1 hour
example.com A	300 s	5 minutes
root NS set	87203 s	of 518400 max

Record	TTL seen	Meaning

4. Notice that the NS and root values shown are partly counted down already. The `com` delegation NS records are published with a TTL of 172800 seconds, which is 48 hours, and the root NS set with 518400 seconds, six days.
5. Standard migration procedure, and this is genuinely what operations teams do, in this order:
    1. At least 48 hours before the move, lower the TTL on the record from its normal value to 60 or 300 seconds. Wait for the old, long TTL to expire everywhere. This waiting step is the part people skip.
    2. Perform the change. Because the TTL is now short, caches drop the old answer within a minute or five.
    3. Watch traffic drain from the old address. Keep the old address serving until it is quiet.
    4. Once stable, raise the TTL back up to reduce query load.
6. The phrase "DNS propagation" is common and slightly wrong. Nothing is pushed anywhere. There is no distribution process to wait for.
7. The honest version: an authoritative change is live the instant the zone is published. What you are waiting for is other people's caches to *expire*. The correct mental model is expiry, not propagation. The practical consequence is the same waiting, but the fix is different: you cannot speed up propagation, but you can plan the TTL in advance.
8. There is one real exception. A change to the NS records must also be made at the parent registry, and registries publish their zones on a schedule, commonly every few minutes for `com`. That step genuinely is a publication delay rather than a cache expiry.
9. Useful commands:

```
dig +noall +answer github.com A          # see the TTL
dig +norecurse @1.1.1.1 github.com A     # cache hit or not
watch -n1 'dig +short +noall +answer github.com A'
sudo dscacheutil -flushcache             # macOS: clear stub cache
sudo killall -HUP mDNSResponder          # macOS: restart resolver
```

10. On macOS both of the last two lines are needed together. Flushing the directory service cache alone does not clear `mDNSResponder`. This has been true since macOS 10.10 and remains the standard pair today.

25.7.6 WORDS - remember these

1. TTL — how many seconds you may reuse an answer — a 32-bit unsigned field per record, counted down by each cache that holds it.
2. Negative caching — remembering that a name does not exist — caching of NXDOMAIN and NODATA responses, bounded by the SOA minimum field, RFC 2308.
3. Cache hit — an answer given without asking anyone — a response served from local cache, identifiable by a TTL below the published maximum.
4. Propagation — the common but inaccurate word for waiting after a change — what is really happening is expiry of previously cached answers.
5. Authoritative TTL — the full value the zone owner publishes — the value seen when querying the authoritative server directly, never counted down.

25.8 Why the reader's machine uses 1.1.1.1

25.8.1 PLAIN - in simple words

1. Your computer has to be told which resolver to use. It cannot guess.
2. Normally it is told automatically, by the router, at the moment it joins the network.

3. The router hands out a small bundle of settings: your address, the gateway to use, and the resolver to use. That bundle is called DHCP.
4. By default the router usually names itself as the resolver, at `192.168.0.1`, and then quietly passes queries on to the internet provider.
5. The reader's Mac is not doing that. It is using `1.1.1.1` directly.
6. That happens for one of a few reasons: somebody set it by hand in network settings, an application set it, a privacy or VPN tool set it, or the provider's own resolver was slow or unreliable and it was changed.
7. The reader's session also showed several `utun` interfaces on the Mac, which are the kind of virtual network interfaces that VPN and tunnel software create. Such software very often changes the resolver as well.
8. That is a strong hint, but it is a hint, not proof. Only the machine's own settings can prove which of those it was.

25.8.2 PLAIN – a picture in your head

1. Think of moving to a new town and needing a doctor.
2. When you register at the town hall, they give you a default doctor: the nearest surgery, chosen for you, no thought needed.
3. Most people keep the default forever, and it is fine.
4. Some people deliberately switch to a different practice: it opens earlier, or has better equipment, or they trust it more with their records.
5. There is a real trade. The default surgery is round the corner. The one you chose might be three towns away.
6. Also, the local surgery knows which local hospital to send you to. The distant one might send you to a hospital near itself instead of near you.

Where this comparison breaks: a doctor is a person with duties to you. A public resolver is a company with no contract with you at all, and its incentives may be quite different from yours. Also you can change resolver in four seconds and change back, whereas changing doctor is paperwork.

25.8.3 PLAIN – a worked example

1. The three best known public resolvers, with real launch dates.

Resolver	Operator	Launched
8.8.8.8	Google	3 Dec 2009
9.9.9.9	Quad9	Nov 2017
1.1.1.1	Cloudflare, APNIC	1 Apr 2018

2. Cloudflare announced `1.1.1.1` on 1 April 2018, which several people initially assumed was a joke because of the date. It was not.
3. The address is not Cloudflare's own. It is held by APNIC, the regional internet registry for Asia and the Pacific, and operated jointly under a research agreement, because `1.1.1.1` had historically been so polluted with stray traffic that nobody could use it.
4. Quad9 was announced in November 2017 by IBM Security, Packet Clearing House and the Global Cyber Alliance. It is now a Swiss foundation based in Zurich, and it blocks known malicious domains by default.
5. Google Public DNS, `8.8.8.8`, was announced on 3 December 2009 and by 2018 was handling over a trillion queries per day.
6. Each also has a second address, so a single failure does not cut you off: `1.0.0.1`, `8.8.4.4`, and `149.112.112.112` respectively.

25.8.4 PLAIN - what is really happening inside

1. When the Mac joins the home network it broadcasts a DHCP request.
2. The router replies with a DHCP offer containing numbered options. Option 3 is the gateway. Option 6 is the list of resolvers.
3. For the reader, option 3 gave `192.168.0.1`, the router. Something then overrode option 6 with `1.1.1.1`.
4. macOS keeps a per-interface resolver list. A manually set value in the network settings for Wi-Fi wins over whatever DHCP offered.
5. A VPN or tunnel client goes further and installs its own resolver settings, often scoped so that some names go one way and others another.
6. Now the important consequence. Your resolver is the thing that talks to authoritative servers. Those servers see the resolver's address, not yours.
7. If your resolver is answering you from Mumbai, then a service that steers traffic by location sees a query from Mumbai, and answers as if you are in Mumbai. That is usually right and occasionally very wrong.

25.8.5 TECHNICAL - the engineer's version

1. The resolver list arrives as DHCP option 6, "Domain Name Server", defined in RFC 2132, March 1997. IPv6 uses a different mechanism, RDNSS in router advertisements, RFC 8106, March 2017.
2. On macOS, inspect what is actually in force with these commands. Note that `/etc/resolv.conf` on macOS is generated and is not the real source of truth.

```
scutil --dns | grep -A2 'resolver #1'
networksetup -getdnsservers Wi-Fi
cat /etc/resolv.conf            # generated, informational only
```

3. Honest trade-offs of using a public resolver instead of the provider's:

Aspect	Provider resolver	Public resolver
Latency to resolver	usually lower	often similar
CDN steering quality	usually better	can be worse
Filtering by ISP	common	depends
Query logging	ISP sees all	operator sees all
Encrypted transport	rare	usually offered

4. The most misunderstood item in that table is CDN steering. Content delivery networks decide which server to give you based on where the *query* appears to come from, which is the resolver, not you.
5. A provider's resolver sits inside the provider's network, and content networks have spent twenty years mapping providers' resolvers to providers' customers. It is a very good signal.
6. A large public resolver breaks that signal unless something replaces it.
7. The replacement is **EDNS Client Subnet**, defined in RFC 7871, published May 2016 as an Informational document. The resolver adds a truncated version of your address, typically the first 24 bits for IPv4, to the query it sends to the authoritative server.
8. That is enough for the content network to place you approximately, and not enough to identify you individually.
9. The honest version, and the RFC says this itself: EDNS Client Subnet is a privacy loss. The RFC recommends it be off by default and used only where the benefit is clear. Google Public DNS sends it to networks that ask. Cloudflare's `1.1.1.1` deliberately does not send it, and instead relies on having its own points of presence very close to users.
10. This is a genuine disagreement between two serious operators. Google's position is that better steering helps users more than the privacy cost. Cloudflare's position is that the privacy cost is not worth paying and that a dense anycast network makes it unnecessary. Both are defensible.

11. For the reader in India this matters concretely. Cloudflare operates points of presence in several Indian cities, so `1.1.1.1` answers from inside India, and the location signal is roughly correct even without client subnet. In a country with fewer points of presence the result can be an answer pointing at a server on another continent.

25.8.6 WORDS - remember these

1. DHCP option 6 — how your machine is told which resolver to use — the Domain Name Server option in a DHCP offer, RFC 2132.
2. Public resolver — a resolver anyone may use — an open recursive resolver operated as a service, such as 1.1.1.1, 8.8.8.8 or 9.9.9.9.
3. EDNS Client Subnet — sending a rough version of your address upstream — an EDNS0 option carrying a truncated client prefix, RFC 7871.
4. CDN steering — choosing which server you are sent to — authoritative responses varied by the apparent location of the querying resolver.
5. scutil — the macOS tool that shows the real resolver configuration — the system configuration utility, queried with `scutil --dns`.

25.9 Why the same name gives different answers

25.9.1 PLAIN - in simple words

1. Ask for `github.com` in India and you may get `20.207.73.82`. Ask in the United States and you may get `140.82.112.3`. Both are correct.
2. There is no single true answer to a DNS question for a large service. There is only the best answer for the asker, right now.
3. Four different mechanisms produce this, and they are constantly mixed up.
4. **Anycast** does it at the routing layer. Everyone gets the same address, and the network delivers them to a different machine.
5. **Geographic steering** does it at the DNS layer. Different people are given different addresses based on where they seem to be.
6. **Latency steering** also does it at the DNS layer, but based on measured speed rather than on a map.
7. **Weighted answers** do it at the DNS layer too, sending a chosen share of people to each of several addresses.
8. The difference matters because you fix them in completely different places.

25.9.2 PLAIN - a picture in your head

1. Two ways to run a chain of shops with one advertised phone number.
2. Way one: print one number on every poster, and arrange with the phone company that calls are routed to the nearest branch automatically. Everyone dials the same digits. That is anycast.
3. Way two: print a different number on the posters in each city. Everyone dials different digits, and each set of digits reaches one specific branch. That is geographic DNS steering.
4. From the customer's side the experience looks identical. From the operator's side they are utterly different problems.
5. With way one, moving a customer to a different branch means changing routing and everyone in a region shifts at once.
6. With way two, moving a customer means changing what the directory says, and the change only reaches people whose cached answer has expired.

Where this comparison breaks: a phone company's routing is a private arrangement with one provider. Anycast routing decisions are made independently by thousands of networks with no coordination, using their own preferences, so the operator cannot fully predict who lands where.

25.9.3 PLAIN – a worked example

1. Here is the difference in one table, using real observed data.

Mechanism	Layer	Who decides	Same reply?
Anycast	Routing	Every network	Yes
Geo steering	DNS	Authoritative server	No
Latency steering	DNS	Authoritative server	No
Weighted round robin	DNS	Authoritative server	No

2. `1.1.1.1` is pure anycast. Everybody in the world is told the same four digits. The routing system decides which of Cloudflare's locations answers.
3. `github.com` uses DNS steering. The reader in India was told `20.207.73.82`. A machine in the United States was told `140.82.112.3` at the same time. Different digits, from the same question.
4. Both were checked and both are real. This is not a fault or an inconsistency.
5. To see it yourself, ask the authoritative server directly and compare with what a distant resolver returns.

```
$ dig +short @1.1.1.1 github.com A      # answers near you
$ dig +short @8.8.8.8 github.com A      # may answer differently
$ dig +short @dns1.p08.nsone.net github.com A
```

25.9.4 PLAIN – what is really happening inside

1. With anycast, one address block is announced into the global routing system from many separate locations.
2. Every router on the internet learns several routes to that block and keeps the one its own policy prefers.
3. Your packet therefore arrives at whichever location the chain of routers between you and it happened to prefer. Nobody consulted a map.
4. Anycast is invisible to DNS. The DNS answer never changes. Two people in different countries can compare notes and see the same address while talking to different machines.
5. With geographic steering, the authoritative name server looks at the address of the resolver that asked, or at the client subnet if one was sent, decides which region that belongs to, and returns a different set of records.
6. This happens before any connection is made. The steering decision is baked into the answer, and it stays baked in for the length of the TTL.
7. That is why steered records almost always carry short TTLs. `github.com` uses 60 seconds. A steering decision that stuck for a day would be useless.
8. Latency steering is the same as geographic steering but the decision comes from continuous measurement rather than from a geography database. It reacts to real network conditions, including congestion.
9. Weighted answers simply divide traffic by proportion, for example 90 percent to a new stack and 10 percent to the old one during a migration.
10. Plain round robin, where a server returns several addresses and rotates the order, is the oldest and dumbest form. It still exists and it still works, but it balances queries, not load, and it has no idea if a server is down.

25.9.5 TECHNICAL – the engineer's version

1. Anycast operates at layer 3 using BGP, the Border Gateway Protocol. The same prefix is originated from multiple sites, usually with the same origin AS number.
2. Failure behaviour is the key property. If a site withdraws its announcement, routers converge on the next best path, typically within seconds to a few tens of seconds, with no DNS change and no TTL to wait for.

3. The classic objection to anycast is that TCP sessions break if routing changes mid-connection. In practice this is rare, because routes are stable for the seconds a typical connection lasts, and operators use techniques such as consistent hashing at the site to limit the damage.
4. DNS-based steering is implemented by authoritative servers that vary answers per query. Common products and services doing this include NS1 Managed DNS, Amazon Route 53 with latency and geolocation routing policies, Akamai's authoritative service, and Azure Traffic Manager.
5. `github.com` is served by NS1 and Route 53 in parallel, both of which support per-query steering, which is consistent with the two very different answers observed in India and in the United States.
6. Route 53 alone offers simple, weighted, latency, failover, geolocation, geoproximity and multivalue answer policies. These are product features, an implementation detail, not standards.
7. The two layers combine. Cloudflare uses anycast for `1.1.1.1` and also returns different answers to different customers for hosted names. Microsoft fronts services with anycast edge addresses and also steers by DNS.
8. Diagnosing which one is in play, in order:

Symptom	Likely mechanism
Same IP everywhere, different rtt	Anycast
Different IP per country	DNS geo steering
IP changes every few minutes	Latency or weighted
Several IPs, order rotates	Round robin

9. The honest version: for a big service it is usually both at once, plus a load balancer inside the destination network doing a third kind of distribution that you cannot see from outside at all. Do not expect a clean single answer.

25.9.6 WORDS - remember these

1. Anycast — one address served from many places — the same BGP prefix originated at multiple sites, with routing selecting the destination.
2. GeoDNS — different answers by location — authoritative responses selected by the apparent geography of the querying resolver or client subnet.
3. Latency steering — answers chosen by measured speed — a routing policy that returns the endpoint with the lowest observed round trip time.
4. Weighted answer — a chosen share of traffic per address — a policy that returns records in proportions set by the operator.
5. Round robin — rotating the order of several addresses — the original and simplest DNS load spreading, which balances queries rather than load.

25.10 The reader's own answer decoded

25.10.1 PLAIN - in simple words

1. The reader asked for `github.com` and got `20.207.73.82`.
2. The first surprise is that this address does not belong to GitHub.
3. It belongs to Microsoft. That sounds wrong until you know one fact.
4. Microsoft bought GitHub. The deal was announced on 4 June 2018 and completed on 26 October 2018, for 7.5 billion United States dollars, paid in Microsoft shares rather than cash.
5. Since then, GitHub has been a Microsoft company, and a lot of GitHub's traffic is carried and fronted by Microsoft's own global network.
6. So a Microsoft address answering for `github.com` is not a mistake and not a hijack. It is the company's own infrastructure.

7. The second thing about this address is where it is. Everything points at India, and specifically at Pune.
8. The reader is in India. So this is the nearby doorway into Microsoft's network, which is exactly what should have been handed out.
9. That makes the reader's fault more interesting, not less. DNS did its job perfectly. The right nearby address was returned. What failed came later.

25.10.2 PLAIN - a picture in your head

1. Imagine a well known bookshop chain that gets bought by a supermarket group.
2. The sign over the door still says the bookshop's name. Customers still call it the bookshop.
3. But the delivery lorries now say the supermarket's name, the warehouses are the supermarket's warehouses, and the loading bay you are sent to is a supermarket depot.
4. If you looked only at the lorry you would think you had the wrong shop.
5. You do not. You have the same shop, reached through a bigger company's logistics network.
6. And the depot you are sent to is the one near your town, not the one near the head office.

Where this comparison breaks: a supermarket owns its lorries outright and you can read the name on the side. On the internet you cannot see ownership directly. You have to look it up in a registry, and the registry records who was *assigned* the address, which is not always who is *using* it today.

25.10.3 PLAIN - a worked example

1. Here is how to check the ownership yourself. Two independent methods, and they should agree.
2. Method one, the registry lookup. Ask the regional internet registry who was assigned this address.

```
whois 20.207.73.82
```

3. The answer, checked in August 2026, contains these fields:

```
NetRange:   20.192.0.0 - 20.255.255.255
CIDR:       20.192.0.0/10
NetName:    MSFT
OrgName:    Microsoft Corporation
Country:    US
RegDate:    2017-10-18
Parent:     NET20 (NET-20-0-0-0-0)
```

4. Method two, the routing lookup. Ask which network is actually announcing this address into the global routing system right now.

```
prefix:  20.192.0.0/10
origin:  AS8075
holder:  MICROSOFT-CORP-MSN-AS-BLOCK, Microsoft Corporation
```

5. The two agree. The address is registered to Microsoft and is announced by Microsoft's own autonomous system number, 8075.
6. Now do the same for the answer a machine in the United States received for the same name, and see the contrast.

Address	Prefix	Announced by
20.207.73.82	20.192.0.0/10	AS8075 Microsoft
140.82.112.3	140.82.112.0/24	AS36459 GitHub

7. Same name, same moment, two different companies' networks. That is section 25.9 in action, and it is the clearest possible demonstration of it.

25.10.4 PLAIN - what is really happening inside

1. Let us be strict about what is proven and what is only suggested, because this is exactly where people overclaim.
2. **Proven.** The address `20.207.73.82` falls inside `20.192.0.0/10`, which the registry records as assigned to Microsoft Corporation, registered on 18 October 2017.
3. **Proven.** That prefix is announced into the global routing system by AS8075, Microsoft's autonomous system.
4. **Proven.** Microsoft acquired GitHub, announced 4 June 2018 and closed 26 October 2018, for 7.5 billion dollars in stock.
5. **Proven from the reader's own trace.** Every router from hop 7 onwards had a hostname ending in `ntwk.msn.net`, which is Microsoft's backbone naming. The final visible hops were `pnq20` and `pnq21`, Pune site codes.
6. **Strongly suggested.** The address is an India-facing edge. A public geolocation database places it in Pune, and the reader's own traceroute reached Microsoft routers in Pune. Two independent signals agree.
7. **Inferred, not proven.** That this specific address belongs to a particular Microsoft edge product. It is consistent with a Microsoft edge front end serving GitHub, but the address publishes no reverse DNS record at all, so there is nothing naming the service.
8. **Not knowable from outside.** Whether GitHub or Microsoft configured this steering, and which internal system chose it.
9. Notice how much can be established with two lookups and a traceroute, and how carefully the last two lines have to be worded.

25.10.5 TECHNICAL - the engineer's version

1. Microsoft holds several very large IPv4 blocks under NetName `MSFT`. Two of them, both registered on 18 October 2017, are:

Block	Range	Size
20.0.0.0/11	20.0.0.0 - 20.31.255.255	2,097,152
20.192.0.0/10	20.192.0.0 - 20.255.255.255	4,194,304

2. The honest version: people say "20.0.0.0/8 is Microsoft". Large parts of it are, and both of the blocks above are, but the statement is loose. Check the specific prefix, not the first octet. The registry answer for the exact address is the only thing worth quoting.
3. IP registry lookups are separate from domain name lookups. For domains, WHOIS was formally retired for generic top level domains on 28 January 2025 in favour of RDAP, the Registration Data Access Protocol. For IP addresses the regional registries still answer WHOIS and also serve RDAP.
4. The RDAP equivalent of the lookup above is a plain HTTPS request to the registry's RDAP endpoint for the address, which returns JSON with the same fields as structured data rather than free text.
5. Reverse DNS on this address returns nothing:

```
$ dig +short -x 20.207.73.82
$
```

6. Compare with GitHub's own address space, which does publish reverse names and even encodes a site code:

```
$ dig +short -x 140.82.112.3
lb-140-82-112-3-iad.github.com.
```

7. `lb` for load balancer and `iad` for Washington Dulles. That naming is a convention, not a standard, but it is a widely used one and it is often the single most useful clue in a trace.

8. Putting the whole picture together for the reader's session:

```
Name asked:      github.com
Answered by:     1.1.1.1 (Cloudflare, anycast, likely in India)
Answer given:    20.207.73.82
Registered to:   Microsoft Corporation (20.192.0.0/10)
Announced by:    AS8075 Microsoft
Path observed:   ISP -> msn.net Delhi -> Mumbai -> Pune -> silence
Reverse DNS:     none published
Conclusion:      correct nearby Microsoft edge for GitHub
```

9. The last line matters for the rest of the reader's investigation. The name resolved, quickly, to a plausible and verifiable address near them. There was never a DNS fault. The failure was in reaching that address, which is a different layer and a different chapter.

25.10.6 WORDS - remember these

1. AS number — the identifier of one network operator — an autonomous system number used in BGP, such as AS8075 for Microsoft.
2. Prefix — a block of addresses written as a range — an address plus a mask length, such as 20.192.0.0/10.
3. RIR — the body that hands out address blocks — a regional internet registry such as ARIN, RIPE NCC or APNIC.
4. RDAP — the modern replacement for WHOIS — the Registration Data Access Protocol, returning JSON over HTTPS, mandatory for gTLDs since Jan 2025.
5. Edge — the nearest point of a big network to you — a front end site that terminates connections close to users and carries traffic onward internally.

25.11 DNS on the wire

25.11.1 PLAIN - in simple words

1. A DNS query is a small message sent over the network, usually as a single packet, and usually answered by a single packet coming back.
2. It goes to port 53. A port is a number that says which service on a machine the message is for.
3. It normally uses UDP, which is the fast, simple way to send a packet: fire and forget, no connection, no guarantee.
4. That choice is why DNS is quick. There is no handshake to set up and tear down for a question that fits in one packet.
5. It also means nothing is confirmed. If the packet is lost, nobody is told. The asker simply waits, gives up, and asks again.
6. If the answer is too big for one packet, the server sets a flag meaning "this was cut short", and the asker tries again over TCP.
7. TCP is the slower, careful way, with a connection and acknowledgements. Same port 53, different transport.
8. Every query carries a random number so the asker can match the answer to the question. That number is where a lot of the trouble in section 25.12 begins.

25.11.2 PLAIN - a picture in your head

1. Think of a postcard versus a registered letter.
2. A postcard is cheap, quick, and you write the reference number of your enquiry on it so the reply can be matched up.
3. Nobody signs for a postcard. If it is lost in the sorting office, you find out only by noticing that no reply came.
4. That is UDP. Most DNS is postcards.

5. A registered letter needs a signature, a receipt, and several exchanges before anything is even delivered. It is much slower and much more certain.
6. That is TCP. DNS uses it only when the reply will not fit on a postcard, or when the whole conversation needs to be private.

Where this comparison breaks: a postcard can be read by anyone handling it, and so can a plain DNS query, which is a fair part of the comparison. But a postcard cannot be forged into your letterbox by a stranger who guessed your reference number, and a DNS answer very much can. That is the attack described in the next section.

25.11.3 PLAIN - a worked example

1. A DNS message has a fixed 12-byte header followed by four sections.

```
+--------------------------------------+
|  Header            12 bytes          |
+--------------------------------------+
|  Question          what was asked    |
+--------------------------------------+
|  Answer            the records asked |
+--------------------------------------+
|  Authority         who is in charge  |
+--------------------------------------+
|  Additional        extras and glue   |
+--------------------------------------+
```

2. In a normal successful reply, the answer section has the records you wanted and the other two are often empty.
3. In a referral, the answer section is empty, the authority section has the NS records of the next level down, and the additional section has their addresses as glue.
4. In a negative reply, the answer section is empty and the authority section has an SOA record, which supplies the negative caching time.
5. Here is a real reply with those parts labelled by `dig` itself.

```
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 51021
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0,
;; ADDITIONAL: 1
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;github.com.                    IN      A
;; ANSWER SECTION:
github.com.             21      IN      A       20.207.73.82
;; Query time: 4 msec
;; SERVER: 1.1.1.1#53(1.1.1.1) (UDP)
;; MSG SIZE  rcvd: 55
```

6. Read the last line. The whole reply was 55 bytes. That is the normal size of DNS traffic, and it is why UDP suits it so well.

25.11.4 PLAIN - what is really happening inside

1. The header is only 12 bytes and every bit of it earns its place.

```
 0                            15
+------------------------------+
|          Query ID            |  16 bits, random
+------------------------------+
|QR|Opcode|AA|TC|RD|RA|Z |RCODE |  flags
```

```
+------------------------------+
|          QDCOUNT             |  questions
+------------------------------+
|          ANCOUNT             |  answers
+------------------------------+
|          NSCOUNT             |  authority records
+------------------------------+
|          ARCOUNT             |  additional records
+------------------------------+
```

2. **Query ID** is a random 16-bit number. The reply must carry the same one or it is discarded.
3. **QR** says query or response. **AA** says the answer came from an authoritative server. **TC** says the reply was truncated.
4. **RD** and **RA** are the recursion flags from section 25.3.
5. **RCODE** is the result code: 0 for success, 2 for server failure, 3 for name does not exist, 5 for refused.
6. Sixteen bits of query ID means 65,536 possible values. Hold that number. It is the whole story of the next section.
7. The original limit on a UDP DNS message was 512 bytes of payload. Anything longer had to be truncated and retried over TCP.
8. That limit was raised by adding a fake record to the additional section that carries options and announces a larger acceptable size. This is EDNS0, and the fake record is called an OPT record. You can see it in the output above, labelled `OPT PSEUDOSECTION`.

25.11.5 TECHNICAL - the engineer's version

1. Port 53 for both UDP and TCP was assigned in RFC 1035. TCP is mandatory to support, not optional, and RFC 7766, March 2016, made that requirement explicit and encouraged persistent TCP connections.
2. TCP is used in four situations: a truncated UDP reply, zone transfers (AXFR and IXFR), any reply the resolver knows will be large, and as the base for DNS over TLS.
3. EDNS0 was introduced in RFC 2671, August 1999, and revised as RFC 6891, April 2013. The OPT pseudo-record advertises the largest UDP payload the sender will accept.
4. Common advertised sizes and why they are argued about:

Size	Where it comes from
512 bytes	Original RFC 1035 limit
1232 bytes	DNS Flag Day 2020 recommendation
1400 bytes	Common older default
4096 bytes	Old BIND default, causes fragments

5. The 1232 figure comes from the smallest IPv6 MTU of 1280 minus 48 bytes of IPv6 and UDP headers. It was recommended by the DNS Flag Day 2020 effort precisely because larger values cause IP fragmentation, and fragmented UDP is dropped by many firewalls and is itself a security problem.
6. Response codes worth memorizing:

RCODE	Name	Meaning
0	NOERROR	Success, or NODATA
1	FORMERR	Malformed query
2	SERVFAIL	Server could not answer
3	NXDOMAIN	Name does not exist
5	REFUSED	Policy refusal

7. SERVFAIL is the one that misleads people. It very often means a DNSSEC validation failure, not a broken server.
8. Name compression, RFC 1035 section 4.1.4, lets a name that has already appeared be replaced by a two-byte pointer. This is why thirteen root servers fit in 512 bytes and why message sizes are hard to predict.
9. The design flaw, stated plainly: the only things binding a reply to a query are the 16-bit ID, the source and destination addresses, the ports, and the question section. In the original design the source port was often fixed. That left roughly 16 bits of entropy for an attacker to guess.

25.11.6 WORDS - remember these

1. Port 53 — the number DNS listens on — the IANA assigned port for DNS over both UDP and TCP.
2. Query ID — the number matching a reply to a question — a 16-bit field that must match, giving only 65,536 possibilities.
3. TC bit — "this reply was cut short" — the truncation flag, telling the client to retry over TCP.
4. EDNS0 — the extension that allows bigger replies and new options — an OPT pseudo-record in the additional section, RFC 6891.
5. SERVFAIL — "I could not answer" — RCODE 2, most often caused by a DNSSEC validation failure rather than an unreachable server.

25.12 DNS security and privacy

25.12.1 PLAIN - in simple words

1. DNS was designed in 1983 for a network of a few hundred trusted machines. Nobody was expected to lie.
2. So the original design has no signatures, no encryption, and no way to tell a genuine answer from a forged one.
3. There are two separate problems, and mixing them up causes bad decisions.
4. Problem one is **integrity**: is this answer true, or did someone forge it?
5. Problem two is **privacy**: who else can see which names I am looking up?
6. The fix for integrity is DNSSEC, which signs records cryptographically.
7. The fix for privacy is encrypting the conversation between you and your resolver, using DNS over TLS or DNS over HTTPS.
8. They are independent. You can have either, both, or neither. DNSSEC does not hide anything, and encryption does not prove anything is true.
9. There is also a third topic which is not a bug at all: DNS is deliberately used to block things, by governments, by companies, and by security teams.

25.12.2 PLAIN - a picture in your head

1. Think of asking a stranger in the street for directions.
2. The integrity problem: how do you know they are not lying and sending you to a pickpocket? A signature you can check is the answer. That is DNSSEC.
3. The privacy problem: everyone standing nearby heard you ask for the address of the bank. Whispering solves that. That is encrypted DNS.
4. Notice that whispering to a liar does not help, and getting a signed note from an honest person shouted across the street does not hide anything.
5. You need both to be safe on both counts, and they are separate purchases.

Where this comparison breaks: a stranger has no reputation and no infrastructure. Real resolvers are large operators with published policies and sometimes with audits. Also, whispering to one person means that one person now knows everything you asked, which is the real trade in encrypted DNS: you have narrowed the audience from many to one, but that one now sees all of it.

25.12.3 PLAIN - a worked example

1. In 2008 Dan Kaminsky found a way to poison a resolver's cache reliably.
2. The idea, simplified: the attacker makes the resolver ask about a name that is certainly not cached, such as `aaa001.example.com`.
3. While the resolver waits, the attacker floods it with forged replies, guessing the 16-bit query ID.
4. The forged reply says "I do not know that name, but here is the address of the name server for the whole of `example.com`, and it is my machine".
5. If one guess lands before the real reply arrives, the resolver caches a poisoned delegation for the entire domain, not just one name.
6. If a guess fails, the attacker simply tries again with `aaa002.example.com`, and can keep trying thousands of times per second.
7. Work the numbers. With 65,536 possible IDs and thousands of attempts per second, a successful poisoning takes seconds to minutes, not centuries.
8. That is what made it serious. Earlier attacks needed luck. This one needed only patience.

25.12.4 PLAIN - what is really happening inside

1. The fix deployed in 2008 was to randomize the source port as well as the ID.
2. The source port is another 16-bit number, so the attacker now has to guess both. That is roughly 32 bits instead of 16.
3. Two to the power of 32 is about 4.3 billion. The same attack that took minutes now takes an infeasibly long time on a normal network.
4. Note that this is a mitigation, not a cure. It makes forgery expensive rather than impossible.
5. The real cure is signing. With DNSSEC, each record set is signed by the zone owner's private key, and the matching public key is vouched for by the parent zone, all the way to the root.
6. The chain looks like this: the root signs a fingerprint of the key for `com`, `com` signs a fingerprint of the key for `github.com`, and `github.com` signs its own records.
7. If you trust the root's key, which is published and widely distributed, you can check every link. A forged answer fails the check and is discarded.
8. Encryption is a separate change entirely. It wraps the query in TLS so that the network between you and your resolver sees only that you contacted the resolver, not what you asked.

25.12.5 TECHNICAL - the engineer's version

1. The Kaminsky flaw is CVE-2008-1447. Vendors released a coordinated patch on 8 July 2008, the first large multi-vendor synchronized DNS patch. Details leaked on 21 July 2008, ahead of the intended embargo, and the full talk was given at Black Hat that August.
2. Source port randomization was formalized in RFC 5452, January 2009, "Measures for Making DNS More Resilient against Forged Answers". Daniel J. Bernstein's djbdns and PowerDNS had already been doing it for years.
3. DNSSEC core specifications are RFC 4033, RFC 4034 and RFC 4035, all published March 2005, replacing the earlier and unworkable RFC 2535 of 1999.
4. The record types added are DNSKEY, RRSIG, DS, NSEC and NSEC3.

Record	Job
DNSKEY	The zone's public key
RRSIG	Signature over one RRset
DS	Hash of a child's key, in parent
NSEC	Proof a name does not exist
NSEC3	Same, with hashed names

5. The root zone was signed on 15 July 2010. The key signing key was rolled for the first time on 11 October 2018.
6. Adoption is genuinely patchy and it is dishonest to pretend otherwise. As of 2025, about 48 percent of country code TLDs operate DNSSEC, but second level adoption is roughly 4 percent in `com` and 5 percent in `net`, while exceeding 50 percent in `nl`, `cz`, `no`, `se` and `nu`.
7. You can see a real DS record for `com` with `dig`:

```
$ dig +noall +answer com DS
com. 53825 IN DS 19718 13 2
  8ACBB0CD28F41250A80A491389424D341522D946B0DA0C029...
```

8. Experts disagree about DNSSEC. Supporters say it is the only real fix for forgery and enables things such as publishing certificate fingerprints in DNS. Critics say the complexity causes more outages than the attacks it prevents, that key management errors take whole domains offline, and that TLS already authenticates the destination. Both positions are held by serious engineers and both have evidence.
9. Encrypted transports:

Name	Port	Standard	Date
DNS over TLS	853	RFC 7858	May 2016
DNS over HTTPS	443	RFC 8484	Oct 2018
Oblivious DoH	443	RFC 9230	Jun 2022

10. DNS over TLS is easy for a network operator to see and to block, because it uses a dedicated port. DNS over HTTPS hides among ordinary web traffic on port 443 and is therefore very hard to distinguish or block.
11. That difference is the whole political argument. Firefox enabled DNS over HTTPS by default for United States users on 25 February 2020 and Chrome followed in May 2020, and network operators, schools and some governments objected loudly, because it removed their visibility and their filtering.
12. Oblivious DNS over HTTPS, RFC 9230, June 2022, splits the knowledge: a proxy sees your address but not your query, and the resolver sees your query but not your address. It is deployed but not widespread.
13. DNS-based filtering and blocking, stated neutrally, comes in three forms:
    1. Refusing to answer, returning NXDOMAIN or REFUSED for a blocked name. Used by Quad9 for malware, by corporate networks, and by regulators.
    2. Answering with a different address, sending users to a warning page.
    3. Sinkholing, where a known malicious name is deliberately pointed at a researcher's machine so infected computers can be counted and contained.
14. NXDOMAIN hijacking is when an internet provider replaces a nonexistent answer with an address of its own, usually showing search results and advertisements. The most famous case was Verisign's Site Finder, which did this for the whole of `com` and `net` from 15 September 2003 and was withdrawn on 4 October 2003 after heavy technical objection.
15. It breaks things beyond the browser. Any software that relies on NXDOMAIN to detect a typo or to finish a search list is confused by a fake address.
16. The most famous sinkhole is the WannaCry kill switch domain, registered on 12 May 2017 by the researcher Marcus Hutchins, which halted the spread of the ransomware because the malware checked whether that name resolved.

25.12.6 WORDS - remember these

1. Cache poisoning — tricking a resolver into storing a false answer — off-path injection of forged responses matching a pending query.

2. Kaminsky attack — the 2008 method that made poisoning practical — repeated forced lookups of random subdomains with forged in-bailiwick delegations.
3. DNSSEC — cryptographic signatures on DNS records — a chain of trust from the root using DNSKEY, RRSIG and DS records, RFC 4033 to 4035.
4. DoT and DoH — encrypting the query to your resolver — DNS over TLS on port 853 and DNS over HTTPS on port 443.
5. Sinkholing — pointing a bad name at a controlled machine — deliberate redirection of malicious domains for containment and measurement.
6. NXDOMAIN hijacking — replacing "no such name" with an advert page — an operator substituting a synthesized A record for a negative response.

25.13 Owning a name

25.13.1 PLAIN - in simple words

1. You cannot buy a domain name. You rent it, for a fixed period, from a chain of organizations.
2. Three roles matter, and they are almost always confused.
3. The **registrant** is you. The person or company the name is registered to.
4. The **registrar** is the shop you buy from. It takes your money, holds your account, and passes your instructions upward.
5. The **registry** runs the whole top level domain. There is exactly one registry per TLD. It holds the actual zone file.
6. So when you register `example.com`, you pay a registrar, and the registrar tells the `com` registry to add your name.
7. Above all of that sits ICANN, which decides which top level domains exist and accredits registrars, and IANA, which maintains the root zone contents.
8. Renting matters. If you stop paying, you lose the name, and someone else can take it. There is no ownership to fall back on.

25.13.2 PLAIN - a picture in your head

1. Think of a market stall pitch in a covered market.
2. The market authority owns the building and decides how many halls there are. That is ICANN.
3. Each hall has one manager who keeps the master plan of every pitch in that hall. That is a registry.
4. You do not talk to the hall manager. You go to one of several letting agents who have a contract with the hall. That is a registrar.
5. You sign with the agent, pay yearly, and your name goes on the hall's master plan. You are the registrant.
6. If you stop paying, the pitch is cleared and relet. Your sign comes down and somebody else's goes up, in the same spot, with your old customers still walking towards it.

Where this comparison breaks: a market stall is physical and somebody would notice a stranger standing in it. A domain can be transferred away silently by someone who gets into your account with the agent, and the first sign is your email and your website quietly arriving somewhere else.

25.13.3 PLAIN - a worked example

1. Here is what actually happens when a name is registered, in order.

```
1. You choose example.com at a registrar.
2. Registrar checks availability with the com registry.
3. You pay, and give contact details.
4. Registrar creates the domain object at the registry,
   using the EPP protocol, with your NS records attached.
5. Registry adds to the com zone:
     example.com.  172800  IN  NS  ns1.yourhost.net.
```

```
     example.com.  172800  IN  NS  ns2.yourhost.net.
6. Registry republishes the com zone (minutes, not days).
7. Your name servers now receive queries for the name.
8. You add A, MX and TXT records at your DNS host.
```

2. Read step 5 again. The registry does not store your website address. It stores only the names of your name servers.
3. That is the delegation. The `com` zone contains hundreds of millions of NS records and almost no A records for customer domains.
4. Step 8 is at a completely different company. Your DNS hosting can be at Cloudflare, Route 53 or NS1 while your registrar is somewhere else entirely.
5. `github.com` shows this split clearly: the delegation at `com` names servers at NS1 and at Amazon Route 53, neither of which is a registrar.

25.13.4 PLAIN - what is really happening inside

1. There are two copies of your name server list, and they must agree.
2. One is at the parent, in the `com` zone, put there by your registrar. This is the one the world actually follows.
3. The other is inside your own zone, as NS records at your apex. This one is what your own servers claim.
4. If they disagree, resolvers follow the parent, and you get confusing half-working behaviour that is hard to debug. Check both when in doubt.
5. When a name expires, it does not vanish that day. There is a sequence of grace periods, and it is worth knowing because it decides whether you can still get your name back.
6. Roughly: the name stops working, then there is a period where you can renew normally, then a redemption period where you can recover it at a much higher fee, then a short pending-delete window, and then it is released.
7. Domain hijacking usually has nothing to do with DNS itself. It is theft of the registrar account, or a fraudulent transfer request, or a compromised email address used for password resets.

25.13.5 TECHNICAL - the engineer's version

1. ICANN, the Internet Corporation for Assigned Names and Numbers, was formed on 30 September 1998 as a California non-profit corporation.
2. The IANA functions, including maintaining the root zone, are performed by Public Technical Identifiers, an ICANN affiliate created when United States government stewardship ended on 1 October 2016.
3. Registrars communicate with registries using EPP, the Extensible Provisioning Protocol, RFC 5730 to RFC 5734, published August 2009.
4. Example registries:

TLD	Registry	Note
com, net	Verisign	Also root zone maintainer
org	Public Interest Registry	Non-profit
uk	Nominet	UK, since 1996
in	NIXI	India

5. The standard gTLD lifecycle, as set by ICANN policy:

Phase	Typical length
Registered term	1 to 10 years

Phase	Typical length
Auto-renew grace	0 to 45 days
Redemption grace	30 days
Pending delete	5 days
Add grace period	5 days

6. The Add Grace Period allows a registrar to delete a new registration for a refund within five days. This was abused at enormous scale around 2007 in a practice called domain tasting, and ICANN largely ended it in 2009 by charging fees on excessive AGP deletions.
7. WHOIS changed fundamentally in 2018. The GDPR took effect on 25 May 2018, and ICANN adopted a Temporary Specification in May 2018 requiring registrars to redact most registrant contact data from public WHOIS output.
8. Before that, a WHOIS query returned a name, postal address, phone number and email for most domains. After it, most gTLD records show only the country, the state, and an anonymized contact form or relay address.
9. WHOIS itself was formally sunset for generic top level domains on 28 January 2025, replaced by RDAP, which returns structured JSON over HTTPS and supports differentiated access for authorized parties.
10. Protection against hijacking is layered, and the important controls are:
    1. EPP status codes such as `clientTransferProhibited` and `clientUpdateProhibited`, set by the registrar on your instruction.
    2. Registry lock, a manual out-of-band process at the registry level, offered for high value names. Changes require human verification.
    3. Two-factor authentication on the registrar account, and a dedicated email address that is not hosted on the domain itself.
11. Real cases worth knowing. On 15 January 2005 the New York provider `panix.com` was transferred away without authorization and its mail and web traffic redirected for most of a day. On 27 August 2013 the Syrian Electronic Army altered DNS delegations for the New York Times and other sites through a compromised reseller of the registrar Melbourne IT.
12. Both were registrar-level compromises. No DNS protocol weakness was involved in either. That is the pattern.

25.13.6 WORDS - remember these

1. Registrant — the person or company a name is registered to — the legal holder of a domain registration for its term.
2. Registrar — the shop you buy the name from — an ICANN-accredited entity that creates and manages domain objects at a registry.
3. Registry — the operator of a whole top level domain — the sole operator of a TLD zone, such as Verisign for `com`.
4. EPP — the protocol registrars use to talk to registries — the Extensible Provisioning Protocol, RFC 5730 to 5734.
5. Redemption grace period — the expensive last chance to get an expired name back — a 30-day ICANN-mandated recovery window before pending delete.
6. Registry lock — the strongest anti-hijack control — a registry-level status requiring manual out-of-band authorization for any change.

25.14 Diagnosing DNS for real

25.14.1 PLAIN - in simple words

1. Most problems blamed on DNS are not DNS. The fastest thing you can do is prove whether it is or not, in one command.
2. The test is simple. Does the name turn into an address, quickly, with a sensible value?
3. If yes, DNS worked, and you must look somewhere else. Stop blaming DNS.
4. If no, then find out where it failed: your own machine, your resolver, or the authoritative servers.
5. The main tool is `dig`. It is precise, it shows everything, and it does not hide anything from you.
6. `nslookup` is older and friendlier and lies to you a little by simplifying. `host` is the short one for a quick answer.
7. On macOS there are two extra commands you need, because macOS caches in a place the others do not touch.
8. The reader's own case is a perfect worked example, and the answer is clean: DNS was never the problem.

25.14.2 PLAIN - a picture in your head

1. Think of a parcel that never arrived.
2. The first question is not "was it stolen". It is "did the address get written on it correctly".
3. If the label on the tracking record shows the right house number, then the address lookup worked and the parcel was lost on the road.
4. If the label shows no address at all, the failure happened at the desk before the parcel ever moved.
5. Those are two entirely different investigations, and doing them in the wrong order wastes hours.
6. In DNS the label is the address `curl` prints before it starts connecting.

Where this comparison breaks: with a parcel you can ask the depot what happened. On the internet, a silent drop tells you nothing at all, and the absence of any reply is itself the most common evidence you will get. You must reason from what did not happen.

25.14.3 PLAIN - a worked example

1. Here is the reader's own session, and the exact line that settles it.

```
$ curl -v https://github.com
*   Trying 20.207.73.82:443...
    ... 15 seconds of nothing ...
* Connection timed out after 15001 milliseconds
```

2. Look at the first line. `curl` printed an IP address.
3. To print that address, `curl` had to resolve `github.com` first. It did, and it did so fast enough that the delay is not visible.
4. Therefore DNS succeeded. Completely. Before the timeout ever started.
5. The 15 seconds of silence happened during the TCP connection attempt, which is a later and separate step.
6. This is the single most useful diagnostic habit in networking: read whether your tool printed a name or a number. A number means resolution finished.
7. For contrast, a genuine DNS failure looks like this, and it fails in under a second rather than after fifteen.

```
$ curl -v https://githubb.com
* Could not resolve host: githubb.com
* Closing connection
curl: (6) Could not resolve host: githubb.com
```

8. Different message, different timing, different layer, different fix.

25.14.4 PLAIN - what is really happening inside

1. Work down the chain, one layer at a time, and stop at the first thing that fails.
2. Layer one, your own cache. Ask twice and see if the answer is instant. If a stale answer is suspected, flush it.
3. Layer two, your configured resolver. Ask it directly by name, with `@`.
4. Layer three, a different resolver. If `1.1.1.1` fails and `8.8.8.8` works, the problem is your resolver or the path to it, not the domain.
5. Layer four, the authoritative servers. Ask them directly. If they answer and your resolver does not, the problem is in between, often DNSSEC.
6. Layer five, the delegation itself. Check that the NS records at the parent match the NS records in the zone.
7. Each of those is one command, and the whole sequence takes under a minute.

25.14.5 TECHNICAL - the engineer's version

1. The `dig` flags that actually matter, with what each is for.

```
dig github.com A                  # the full, normal query
dig +short github.com             # just the answer, for scripts
dig +noall +answer github.com     # answer section only, with TTL
dig @1.1.1.1 github.com           # ask a specific resolver
dig @dns1.p08.nsone.net github.com  # ask the authoritative server
dig +trace github.com             # walk the whole delegation chain
dig +norecurse @1.1.1.1 github.com  # cache hit test, RD cleared
dig +dnssec github.com            # request signatures, see AD flag
dig +tcp github.com               # force TCP instead of UDP
dig -x 20.207.73.82               # reverse lookup
dig github.com ANY                # mostly useless now, see below
dig +nssearch github.com          # find and query all its NS
```

2. `+trace` is the one to reach for when a delegation looks wrong. It shows every referral, so you can see exactly which level gives the bad answer.
3. `+norecurse` against a recursive resolver is the cache test. If you get a full answer with the RD bit clear, it was cached. If you get nothing, it was not.
4. `dig ANY` is largely dead. RFC 8482, January 2019, allows servers to return a minimal response to ANY queries, and Cloudflare and others do exactly that, because ANY was heavily used for amplification attacks.
5. The other tools:

Tool	Best for	Caution
dig	Everything, precisely	Verbose by default
host	One quick answer	Hides detail
nslookup	Interactive, on Windows	Own resolver logic
dscacheutil	macOS cache view	Not the whole cache
scutil –dns	Real macOS config	Long output

6. `nslookup` deserves the warning. It has its own resolution behaviour and does not always use the system resolver the way applications do, so an answer from `nslookup` is not proof that an application will get the same.
7. macOS cache handling, and both lines are needed:

```
sudo dscacheutil -flushcache
sudo killall -HUP mDNSResponder
```

8. A single command that separates DNS from connectivity, because it does the two steps in order and reports each one:

```
dig +short github.com && \
  nc -vz -w 5 $(dig +short github.com | head -1) 443
```

9. If the first part prints an address and the second part times out, DNS is fine and the transport is not. That is precisely the reader's situation.
10. Applied to the reader's session, stated as findings:
    1. `github.com` resolved to `20.207.73.82`. Resolution succeeded.
    2. The address is verifiably Microsoft's, registered and announced by AS8075, and geolocates to Pune, near the reader.
    3. `curl` printed the address and then received nothing for 15 seconds.
    4. The traceroute reached Microsoft routers in Pune and then stopped.
    5. The same request succeeded immediately over mobile data.
11. Therefore DNS worked perfectly and can be excluded with confidence. The fault lies on the path between the reader's fixed line and that address.
12. Be careful about the last step. The evidence supports a silent drop somewhere on that path. It does not identify who dropped it, and it does not support saying the server was down, because the same address answered fine over a different access network minutes later.

25.14.6 WORDS - remember these

1. dig — the standard DNS query tool — domain information groper, from BIND, showing the full message including flags and timings.
2. `+trace` — walk the tree yourself — dig mode that issues iterative queries from the root and prints every referral.
3. `+short` — answer only — dig output mode giving bare record data, suitable for scripts.
4. `+norecurse` — ask without asking for work — clears the RD bit, used to test whether a resolver already holds an answer in cache.
5. mDNSResponder — the macOS resolver service — the system daemon handling both unicast DNS caching and Multicast DNS.

25.98 Common wrong ideas

1. Wrong: my computer walks the DNS tree and asks the root servers. Right: your computer asks one resolver, once. The resolver walks the tree. Your machine almost never speaks to a root server in its whole life.
2. Wrong: there are thirteen root servers. Right: there are thirteen root server *names* and addresses, answered by over 1,900 physical instances worldwide, 2,003 of them on 13 August 2026.
3. Wrong: a DNS change has to propagate around the world. Right: nothing is pushed anywhere. The change is live immediately at the authoritative servers. You are waiting for other people's caches to expire.
4. Wrong: TTL guarantees how long an answer is kept. Right: TTL is a maximum. A cache may drop an answer sooner, and some browsers and applications ignore short TTLs entirely.
5. Wrong: `www.example.com` and `example.com` are the same name. Right: they are two different names. They often point at the same place, but only because somebody configured records for both.
6. Wrong: I can put a CNAME at the top of my domain. Right: you cannot. The apex must hold SOA and NS records, and RFC 1034 forbids any other data alongside a CNAME. The alternatives are provider features, not standards.
7. Wrong: DNSSEC encrypts my DNS queries. Right: DNSSEC signs answers so forgery can be detected. It hides nothing. Encryption is a separate thing, DNS over TLS or DNS over HTTPS.

8. Wrong: using 1.1.1.1 always makes browsing faster. Right: it usually makes lookups fast, but it can give you a content server further away than your provider's resolver would have, because location steering is based on the resolver, not on you.
9. Wrong: the reverse lookup of an address should give back the name you started with. Right: forward and reverse are controlled by different parties and often disagree, and many addresses publish no reverse record at all. The reader's own `20.207.73.82` publishes none.
10. Wrong: a SERVFAIL means the server is down. Right: it means the resolver could not produce an answer, and by far the most common cause today is a DNSSEC validation failure.

25.99 Chapter summary in 20 lines

1. DNS turns names people remember into numbers machines route to.
2. Before it, one hand-edited file, `HOSTS.TXT`, was copied by every machine from Stanford Research Institute, and it could not scale.
3. Paul Mockapetris designed DNS in 1983, published as RFC 882 and RFC 883, replaced in November 1987 by RFC 1034 and RFC 1035, still the base today.
4. Names are a tree read right to left, from an invisible root written as a single dot, through top level domains, down to whatever you create.
5. Four participants: a stub resolver in your machine, a recursive resolver that does all the work, and authoritative servers in three tiers.
6. Your machine asks once and waits. The recursive resolver does the running around. That single sentence prevents most confusion about DNS.
7. A cold lookup of `github.com` takes about eight packets and roughly 90 milliseconds; a warm one takes two packets and under 10 milliseconds.
8. Thirteen root names, A to M, run by twelve organizations, served by 2,003 instances as of 13 August 2026, using anycast. Thirteen was a 512-byte limit.
9. Record types each answer a different question: A and AAAA for addresses, CNAME for aliases, MX for mail, NS for delegation, SOA for zone settings.
10. A CNAME cannot sit at the apex of a domain, because SOA and NS must be there and a CNAME excludes all other data. This catches everyone once.
11. Caching is what makes DNS survive. Every answer carries a TTL, which is a promise of a maximum reuse time, not a guarantee.
12. Lower the TTL at least two days before a migration, make the change, then raise it again. There is no propagation, only expiry.
13. Your resolver normally arrives by DHCP option 6 from the router. The reader overrode this and uses `1.1.1.1`, launched by Cloudflare on 1 April 2018.
14. Public resolvers trade local knowledge for scale and privacy. EDNS Client Subnet, RFC 7871, partly restores the location signal, at a privacy cost.
15. Different people get different answers for two separate reasons: anycast at the routing layer, and geographic or latency steering at the DNS layer.
16. The reader's `github.com` resolved to `20.207.73.82`, inside `20.192.0.0/10`, registered to Microsoft and announced by AS8075.
17. That is correct: Microsoft acquired GitHub for 7.5 billion dollars, announced 4 June 2018 and closed 26 October 2018, and fronts its traffic.
18. On the wire DNS is UDP port 53, a 12-byte header and four sections, with a 16-bit query ID that made the 2008 Kaminsky cache poisoning attack possible.
19. DNSSEC signs answers, from the root signed on 15 July 2010; DNS over TLS on port 853 and DNS over HTTPS on port 443 encrypt them. They are separate.
20. In the reader's own fault, DNS worked perfectly: the name resolved fast to a correct nearby address, so the failure was definitely somewhere later.

26

CHAPTER

KB

Packets and Routing

26.0 What this chapter gives you

1. You will be able to explain why data is chopped into small pieces at all, instead of being sent as one long stream, and name the four separate reasons.
2. You will be able to draw a real packet, layer by layer, with byte counts, and work out how much of the wire is spent on wrapping rather than content.
3. You will be able to read every field of an IPv4 header, say what each one is for, and decode a real header from the reader's laptop by hand.
4. You will be able to explain what happens when a packet is too big for a link, what IPv4 does, what IPv6 refuses to do, and why a filtered ICMP message can make a connection die in silence.
5. You will be able to list, in order, the eight things a router does to a packet, and say which addresses change at every hop and which never change.
6. You will be able to read a real routing table, apply longest prefix match to a real address, and explain the reader's default route through 192.168.0.1.
7. You will be able to say how routers learn routes, from static lines typed by hand up to BGP, and explain what convergence is and why it is not instant.
8. You will be able to explain autonomous systems, transit, peering and internet exchanges, look up which network owns an address, and read AS8075 correctly.
9. You will be able to explain why BGP is about money and policy rather than distance, and describe three real incidents where it went badly wrong.
10. You will be able to explain why the reader can see twelve hops outward and still know nothing at all about the way back.

26.1 Why data is split up at all

26.1.1 PLAIN – in simple words

1. When your computer sends something big, it does not send it as one lump.
2. It cuts the data into many small pieces and sends the pieces one after another.
3. Each piece is called a **packet**. A packet is a small parcel of data with a label on the front saying where it is going.
4. A typical packet on a home network carries about 1,500 bytes at most.
5. There are four reasons for doing it this way, and all four matter.
6. Reason one is fairness. A wire is shared. If one person sent a whole film in one unbroken lump, everyone else would wait until it finished.
7. Reason two is the cost of mistakes. If something goes wrong, you only resend the small piece that was damaged, not the whole file.
8. Reason three is memory. A machine in the middle only needs room to hold a few small pieces at a time, not the whole file.

9. Reason four is mixing. Small pieces from many different conversations can be interleaved on one wire, so many things happen at once.

26.1.2 PLAIN - a picture in your head

1. Think about how you would move a house full of furniture across a country.
2. You would pack it into boxes. Each box has a label with the new address.
3. The boxes travel separately. Some go on one lorry, some on another.
4. If one box is dropped, you replace the contents of that box only. The other boxes are unaffected.
5. Other people's boxes ride on the same lorries at the same time. The lorry is never held for one customer.
6. This is exactly what packets do, and it is why the postal system is a much better picture of a network than a pipe is.
7. A pipe suggests a continuous stream that arrives in order, complete, with nothing else in it. None of that is true of a network.
8. Where this comparison breaks: real parcels are almost never lost or reordered, and the post office never sends two boxes by two different routes on purpose. A network does all three, routinely, every second.

26.1.3 PLAIN - a worked example

1. Suppose two people share one 100 megabit per second link.
2. Person A wants to send a one gigabyte file. Person B wants to load a web page of 50 kilobytes.
3. One gigabyte is 8,000,000,000 bits. At 100 megabits per second that takes 80 seconds to push onto the wire.
4. If the link carried whole files one at a time, person B would wait 80 seconds for a page that needs half a second of transfer.
5. With packets, the link carries a piece from A, then a piece from B, then a piece from A, and so on. Person B finishes in about a second.
6. Now the error case. Assume 1 packet in 1,000 is lost.
7. Sent as packets of 1,448 bytes each, the file is 690,608 packets, and one lost packet costs 1,448 bytes, which is about 0.116 milliseconds of link time.

Way of sending	Cost of one loss	Wait for other user
One 1 GB lump	1 GB resent, 80 s	Up to 80 s
1,448 B packets	1,448 B resent	About 0.12 ms

26.1.4 PLAIN - what is really happening inside

1. There are two old ways to share a network, and they are worth naming clearly.
2. The first is **circuit switching**. Before you send anything, the network reserves a path from end to end and holds it for you until you hang up.
3. The classic telephone call worked this way. When you dialled, exchanges along the way set up a chain of connections, and that chain was yours alone.
4. A voice circuit was fixed at 64,000 bits per second. That capacity was yours whether you were talking or silent.
5. The bad part: silence is wasted. Two people on a call are silent about half the time, and that half of the capacity is thrown away.
6. The worse part: setup takes time, and if any link on the reserved path fails, the call drops. There is no way to slip round the break.
7. The second way is **packet switching**. Nothing is reserved. Every packet carries its own destination address and is dealt with on arrival.
8. A machine in the middle looks at the address, picks an outgoing link, and sends it. If the link is busy, the packet waits briefly in a queue.

9. Nothing about the earlier packets is remembered. Each packet is judged on its own. That property is called being **stateless**.
10. The bad part: delay is not steady. A packet may wait behind others. Under heavy load, packets are dropped rather than delayed forever.

26.1.5 TECHNICAL - the engineer's version

1. Packet switching was invented twice, independently, in the early 1960s.
2. Leonard Kleinrock published queueing analysis of message flow in large communication networks at MIT in 1961, later expanded into his 1962 thesis.
3. Paul Baran at the RAND Corporation published the eleven-volume report On Distributed Communications in August 1964, describing what he called distributed adaptive message block switching, designed to survive damage.
4. Donald Davies at the National Physical Laboratory in the United Kingdom reached the same design in 1965 and gave it the name that stuck: **packet**.
5. The ARPANET, the first large working packet network, carried its first message on 29 October 1969 from UCLA to the Stanford Research Institute.
6. Vint Cerf and Robert Kahn published A Protocol for Packet Network Intercommunication in IEEE Transactions on Communications in May 1974, which is the direct ancestor of TCP and IP.
7. The ARPANET switched from the older NCP to TCP/IP on 1 January 1983, a single cut-over now remembered as flag day.
8. Serialization delay is the time to clock a frame onto the wire, and it is exactly frame bits divided by link rate.

Link rate	1,518 B frame	64 B frame
10 Mbit/s	1,214 us	51.2 us
100 Mbit/s	121.4 us	5.12 us
1 Gbit/s	12.14 us	0.512 us
10 Gbit/s	1.214 us	0.0512 us

9. Tools: `ping -s` varies packet size, `iperf3` measures throughput, and `tc qdisc` on Linux shows the queue behaviour underneath.

26.1.6 WORDS - remember these

1. Packet — a small labelled parcel of data — a protocol data unit carrying a header and a payload, forwarded independently.
2. Circuit switching — reserve a path first, then talk — connection-oriented networking with resources committed for the call duration.
3. Packet switching — no reservation, address on every parcel — connectionless store-and-forward networking with per-packet forwarding decisions.
4. Multiplexing — many conversations sharing one wire — statistical multiplexing of independent flows onto a shared link.
5. Serialization delay — the time to push a packet onto the wire — frame size in bits divided by the link rate in bits per second.

26.2 What a packet actually is

26.2.1 PLAIN - in simple words

1. Every packet has exactly two parts: a **header** and a **payload**.
2. The header is the label on the outside. It says where the packet is going, where it came from, and what kind of thing is inside.

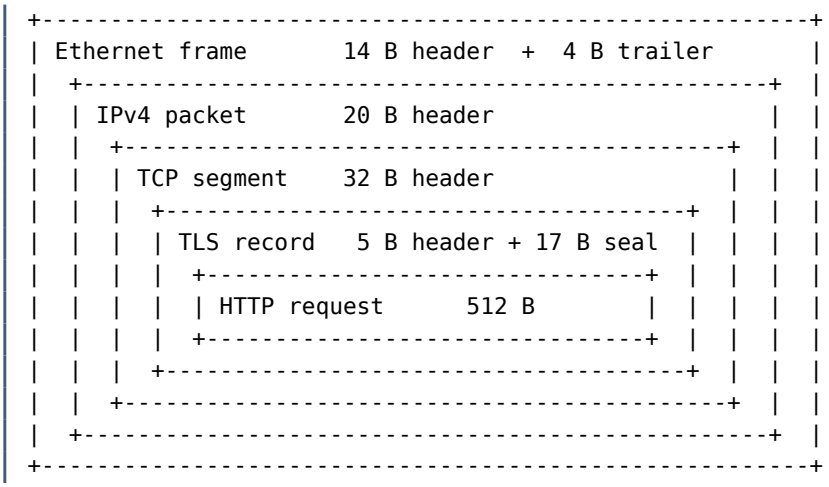
3. The payload is the thing being carried. The network does not read it.
4. Here is the part that surprises people: the payload of one packet is usually another whole packet, with its own header.
5. The outermost label is used by the local wire. The next one in is used by the routers. The next one in is used by the two computers at the ends.
6. The next one in is used by the encryption layer. The innermost one is the thing you actually wanted, like a web request.
7. Putting a box inside a box is called **encapsulation**.
8. Taking the boxes apart again at the far end is called **decapsulation**.

26.2.2 PLAIN - a picture in your head

1. Think of a diplomatic pouch being sent between two countries.
2. Inside the outer canvas sack there is a sealed courier bag.
3. Inside the envelope there is a single sheet of paper with the message.
4. The airline handles the sack. It reads the sack's tag and nothing else.
5. The courier company handles the bag. It reads the bag's tag and nothing else.
6. The embassy opens the case. The ambassador opens the envelope. Only the ambassador reads the paper.
7. Where this comparison breaks: in a real network the outer wrapper is thrown away and rebuilt at every single hop, not carried end to end. The airline tag is replaced at every airport. That is section 26.5.

26.2.3 PLAIN - a worked example

1. Let us draw one real packet: the reader's laptop sending an HTTPS request to `github.com` at `20.207.73.82`.
2. Assume the request itself, the plain text a web server would read, is 512 bytes.



3. Now add it up from the inside out.

Layer	Adds	Running total
HTTP request	512 B	512 B
TLS 1.3 record	22 B	534 B
TCP segment	32 B	566 B
IPv4 packet	20 B	586 B
Ethernet frame	18 B	604 B

4. The TCP 32 bytes are the fixed 20 byte header plus 12 bytes of options, which on a modern machine are two padding bytes and a timestamp.

5. So 604 bytes go on the wire to deliver 512 useful bytes.
6. The overhead is 604 minus 512, which is 92 bytes, or 15.2 percent.

26.2.4 PLAIN - what is really happening inside

1. Encapsulation happens in a fixed order on the way out, top to bottom.
2. The browser writes the HTTP request bytes and hands them to the TLS layer.
3. TCP cuts the stream into segments no bigger than the agreed maximum, puts its header on each one, and hands each segment to IP.
4. IP puts its own 20 byte header on the front, with the source and destination addresses, and hands the packet to the network card driver.
5. The driver wraps it in an Ethernet or Wi-Fi frame with hardware addresses, and the card turns the bits into signals.
6. Decapsulation at the far end is the same list read backwards.
7. IP checks the destination is itself, strips the IP header, reads the protocol number, and hands the rest to TCP.
8. TCP checks the port number, strips its header, puts the bytes back in order, and hands a byte stream to TLS.

26.2.5 TECHNICAL - the engineer's version

1. Ethernet II framing carries a 6 byte destination MAC, a 6 byte source MAC and a 2 byte EtherType, followed by payload and a 4 byte frame check sequence.
2. EtherType `0x0800` means IPv4, `0x86DD` means IPv6 and `0x0806` means ARP. These are assigned by the IEEE Registration Authority.
3. The standard Ethernet payload maximum is 1,500 bytes, so the standard frame is 1,518 bytes including the check sequence, and the wire slot is 1,538.
4. On 802.11 Wi-Fi the picture differs: a data frame header is 24 to 30 bytes plus an 8 byte LLC/SNAP shim, so the same IP packet costs more.
5. The reader is on home broadband over a router at `192.168.0.1`. If the laptop is on Wi-Fi, the first hop uses 802.11 framing and only the wired hops beyond the router use Ethernet framing. The IP packet inside is identical.
6. TLS 1.3, specified in RFC 8446 from August 2018, adds 5 bytes of record header, 1 byte of inner content type and a 16 byte AEAD tag per record.

Packet kind	Useful bytes	Frame bytes	Overhead
Pure TCP ack	0	70	100 percent
Small request	512	604	15.2 percent
Full segment	1,426	1,518	6.1 percent

7. Tools: `tcpdump -e -x` prints the frame header and the raw bytes; Wireshark shows the same nesting as a tree you can expand layer by layer.

26.2.6 WORDS - remember these

1. Header — the label on the front — the fixed-format control fields a protocol layer prepends to the data it is carrying.
2. Payload — the thing being carried — the opaque body of a protocol data unit, normally the entire packet of the layer above.
3. Encapsulation — putting a box inside a box — wrapping an upper-layer PDU as the payload of a lower-layer PDU on transmission.
4. Decapsulation — unwrapping — stripping a header and demultiplexing the payload to the correct upper-layer protocol on reception.

5. Frame — the outermost wrapper on one wire — a link-layer PDU with hardware addresses, valid only between two directly connected devices.

26.3 The IPv4 header field by field

26.3.1 PLAIN - in simple words

1. The IP header is the only part of the packet that routers in the middle read.
2. It is 20 bytes long in almost every packet you will ever see.
3. The two most important fields are the source address and the destination address. Together they take 8 of the 20 bytes.
4. The next most important is a countdown number that stops packets circling forever if the routers get confused.
5. There is a number saying what is inside, so the receiver knows whether to hand the payload to TCP, to UDP, or to something else.
6. The rest is machinery for cutting a packet into smaller pieces, plus a couple of bits for marking traffic as more or less urgent.

26.3.2 PLAIN - a picture in your head

1. Picture a customs declaration form stuck to the outside of a parcel.
2. Line one says which version of the form this is, so the clerk knows the layout.
3. Line three says how urgent the parcel is and whether the depot was crowded.
4. Lines five to seven exist only for parcels that had to be split: a batch number, a flag saying whether more pieces follow, and where this piece sits.
5. Line eight is a stamp saying how many more depots this parcel may pass through before being thrown away.
6. Line nine says what is inside so the right department gets it.
7. Where this comparison breaks: a customs form is read once. An IP header is read, altered and re-stamped at every single depot, and the checksum has to be recomputed each time.

26.3.3 PLAIN - a worked example

1. Here is the IP header of the first packet the reader's laptop sent towards `github.com`, the TCP SYN that opens the connection to port 443.
2. The laptop's own address on the home network is not recorded in the session notes, so we write it as `192.168.0.14`. Treat that one number as a stand-in. Every other value here is exactly what such a packet carries.

```
0000   45 00 00 40 4a 3b 40 00 40 06 d1 a5 c0 a8 00 0e
0010   14 cf 49 52
```

3. Read it byte by byte.

Bytes	Value	Meaning
45	4, 5	IPv4, header is 20 bytes
00	0, 0	No priority, no congestion
00 40	64	Total length 64 bytes
4a 3b	19003	Identification
40 00	DF set	Do not fragment, offset 0
40	64	TTL 64 hops left
06	6	Payload is TCP
d1 a5	0xd1a5	Header checksum

Bytes	Value	Meaning
c0 a8 00 0e	192.168.0.14	Source
14 cf 49 52	20.207.73.82	Destination

4. Total length 64 makes sense: a 20 byte IP header plus a 44 byte TCP header, because a SYN carries options and no data.
5. TTL 64 is the macOS and Linux default. Windows uses 128. Many routers use 255 for their own packets. These are conventions, not requirements.
6. When the reader's router at `192.168.0.1` forwards this packet, TTL drops to 63 and the checksum becomes `0xd2a5`.

26.3.4 PLAIN - what is really happening inside

1. The checksum is a one's complement sum, and it is worth seeing once.
2. Split the header into ten 16 bit words, with the checksum field set to zero.
3. Add them all. Whenever the total goes past 16 bits, wrap the carry round and add it back at the bottom.
4. Flip every bit of the result. That is the checksum you store.
5. For the header above the running total is `0x2e5a`, and flipping every bit gives `0xd1a5`, which is exactly what is in the packet.
6. A receiver checks it by summing all ten words including the checksum. If the header is intact the answer is always `0xffff`.
7. This checksum covers the header only, not the payload. TCP and UDP have their own checksums for the data.
8. That split is intentional. Routers change header fields at every hop, so the header checksum must be recomputed at every hop, and it is kept cheap.

26.3.5 TECHNICAL - the engineer's version

1. The IPv4 header is defined in RFC 791, Internet Protocol, September 1981, and its layout has not changed since.

Field	Bits	Purpose
Version	4	Always 4
IHL	4	Header length in 32-bit words
DSCP	6	Class-of-service marking
ECN	2	Congestion experienced signal
Total Length	16	Header plus payload, in bytes
Identification	16	Groups fragments of one datagram
Flags	3	Reserved, DF, MF
Fragment Offset	13	Position, in 8-byte units
TTL	8	Hop budget
Protocol	8	Next header type
Header Checksum	16	One's complement, header only
Source Address	32	Sender IPv4 address
Destination Address	32	Target IPv4 address
Options	0 to 320	Rare, usually absent

2. IHL is measured in 32 bit words, so the minimum legal value is 5, meaning 20 bytes, and the maximum is 15, meaning 60 bytes.
3. The 8 bits after version and IHL were originally a Type of Service byte. RFC 2474, December 1998, redefined the top 6 bits as the Differentiated Services Code Point, and RFC 3168, September 2001, took the bottom 2 for Explicit Congestion Notification.

4. Total Length is 16 bits, so an IPv4 datagram cannot exceed 65,535 bytes.
5. RFC 6864, February 2013, updated the Identification field: it is only meaningful when a datagram is actually fragmented, and senders that always set DF may leave it as anything, including zero.
6. Common Protocol numbers, assigned by IANA: 1 is ICMP, 6 is TCP, 17 is UDP, 41 is IPv6 encapsulation, 47 is GRE, 50 is ESP, 89 is OSPF, 132 is SCTP.
7. RFC 1812, Requirements for IP Version 4 Routers, June 1995, is the document that tells a router exactly how to treat each of these fields.
8. Tools: `tcpdump -vvn -x host 20.207.73.82` prints the decoded fields and the raw hex; in Wireshark the display filter `ip.ttl < 5` finds nearly expired packets, and `ip.flags.df == 1` finds Do Not Fragment packets.

26.3.6 WORDS - remember these

1. TTL — a countdown that stops packets circling forever — time to live, an 8 bit hop budget decremented by every forwarding router.
2. DF bit — do not cut this packet up — the Don't Fragment flag, which forces a router to drop rather than fragment an oversized packet.
3. Protocol number — a label saying what is inside — an 8 bit IANA-assigned value selecting the next-layer protocol.
4. Header checksum — a small sum that catches corruption — a 16 bit one's complement checksum over the IPv4 header only.
5. DSCP — a priority marking — the 6 bit Differentiated Services Code Point from RFC 2474.

26.4 Fragmentation and MTU

26.4.1 PLAIN - in simple words

1. Every link has a limit on how big a single packet may be.
2. That limit is called the **MTU**, short for maximum transmission unit.
3. A packet travels over many links in a row, and they need not all have the same limit.
4. So a packet can start out legal, travel happily for six hops, and then arrive at a link where it is too big.
5. There are only three things a router can do with a packet that is too big.
6. It can cut the packet into smaller pieces and send them separately.
7. It can throw the packet away and send a complaint back to the sender saying "too big, the limit here is this number".
8. Or it can throw the packet away and say nothing at all.

26.4.2 PLAIN - a picture in your head

1. Imagine driving a lorry across a country with a series of low bridges.
2. Option one: the bridge keeper unloads your lorry into three smaller vans and sends them on. Slower, more paperwork, but the goods arrive.
3. Option two: the bridge keeper turns you back and hands you a note saying "maximum height here is two point five metres". You repack and try again.
4. Option three: the bridge keeper waves you into a ditch and goes back to reading the newspaper. You are simply never seen again.
5. Where this comparison breaks: a lorry driver knows immediately that they hit a bridge. A packet sender has no way to know unless somebody tells it.

26.4.3 PLAIN - a worked example

1. Take a 1,500 byte IPv4 packet: 20 bytes of header and 1,480 bytes of data.
2. It reaches a link with an MTU of 1,400 bytes, and the DF bit is not set, so the router is allowed to cut it up.

3. Every fragment except the last must be a multiple of 8 bytes, because the offset field counts in 8 byte units.
4. The biggest data chunk that fits in 1,400 bytes is 1,380, and the biggest multiple of 8 not above 1,380 is 1,376.

Fragment	Offset field	Data bytes	MF flag
1	0	1,376	1
2	172	104	0

5. Offset 172 means 172 times 8, which is 1,376 bytes into the original data. That is exactly where fragment 1 stopped.
6. 1,376 plus 104 is 1,480, which is the original data. Nothing is lost.

26.4.4 PLAIN – what is really happening inside

1. IPv4 lets any router on the path fragment a packet. IPv6 does not.
2. IPv6 made a deliberate decision: routers never fragment. If a packet is too big, the router drops it and sends back a "packet too big" message.
3. Modern IPv4 senders behave the same way voluntarily. They set the DF bit on every packet, which forbids any router from cutting it up.
4. That means the sender must find out for itself how big it may go. The method is called **path MTU discovery**.
5. It works like this. Send a full-size packet with DF set. If a link on the way cannot take it, that router drops it and sends back an ICMP message saying "fragmentation needed" together with the MTU it can accept.
6. The sender reads that number, shrinks its packets to fit, and retransmits.
7. The whole scheme depends on one thing: the ICMP message must get back.
8. When that happens, the big packets vanish and no message comes back. The sender keeps retransmitting the same too-big packet forever.
9. This is called a **path MTU black hole**, and it produces a very particular symptom: small things work, big things hang.

26.4.5 TECHNICAL – the engineer's version

1. Path MTU discovery for IPv4 is RFC 1191, November 1990. For IPv6 it is RFC 8201, July 2017, which obsoleted RFC 1981.
2. The IPv4 signal is ICMP type 3, code 4, Destination Unreachable with Fragmentation Needed and DF set. The next-hop MTU is carried in the message.
3. The IPv6 signal is ICMPv6 type 2, Packet Too Big. IPv6 forbids router fragmentation entirely, by RFC 8200, so the sender must react or fail.
4. IPv6 sets a hard floor: every link must support at least 1,280 bytes. IPv4's equivalent floor is 68 bytes to forward and 576 bytes to reassemble.

Link type	Typical MTU	Note
Ethernet	1,500	The default everywhere
PPPoE DSL	1,492	8 bytes of PPPoE header
IPsec tunnel	1,400 to 1,438	Depends on ciphers
WireGuard	1,420	Common default
IPv6 minimum	1,280	Floor set by RFC 8200
Jumbo frames	9,000	Data centre convention

5. Because ICMP filtering is so common, the IETF specified a method that does not need ICMP at all: Packetization Layer Path MTU Discovery, RFC 4821, March 2007, extended to datagram transports in RFC 8899, September 2020.
6. A quick manual probe on macOS, which sends a 1,472 byte payload plus 8 bytes of ICMP header plus 20 bytes of IP header, exactly 1,500 bytes:

```
ping -D -s 1472 20.207.73.82
ping -D -s 1400 20.207.73.82
ping -D -s 1200 20.207.73.82
```

7. If the 1,472 probe fails and the 1,200 probe succeeds, you have found an MTU restriction on the path. On Linux the flag is `-M do` instead of `-D`.
8. Tunnels are the usual cause, and the reader's machine showed several `utun` interfaces, which on macOS are tunnel devices. A tunnel always reduces the usable MTU.
9. Now the honest part, because it matters. A path MTU black hole is a genuine and common cause of exactly the kind of silent failure the reader saw, and it belongs on the candidate list.
10. But the evidence here points elsewhere. `curl` printed `Trying 20.207.73.82:443...` and then nothing for 15 seconds. `curl` prints `Connected to` only after the TCP handshake completes, and it never did.
11. So the packet that got no answer was the SYN, which is 64 bytes. An MTU problem does not stop a 64 byte packet.
12. What is PROVEN: the first small packet of the connection received no reply of any kind. What is merely SUGGESTED: something on that path silently discarded it.
13. The classic MTU black hole signature is the opposite shape: handshake succeeds, small responses arrive, and the connection stalls on the first full-size packet. That is not what happened.

26.4.6 WORDS - remember these

1. MTU — the biggest packet a link will carry — maximum transmission unit in bytes, 1,500 for standard Ethernet.
2. Path MTU — the smallest limit anywhere along the route — the minimum MTU over all links on a given path.
3. Fragmentation — cutting a packet into smaller ones — splitting an IPv4 datagram across multiple packets sharing an identification value.
4. Path MTU discovery — asking the network how big you may go — RFC 1191 for IPv4 and RFC 8201 for IPv6, using ICMP too-big messages.
5. Black hole — packets vanish and nobody says why — a path where oversized packets are dropped and the ICMP notification is filtered.

26.5 What a router actually does, mechanically

26.5.1 PLAIN - in simple words

1. A router is a machine with two or more network connections that passes packets from one to another.
2. Step one: a frame arrives on a wire and the router checks it is not damaged.
3. Step two: it throws away the outer wrapper, the one with hardware addresses, because that wrapper was only ever for that one wire.
4. Step three: it reads the destination IP address inside.
5. Step four: it looks that address up in its routing table to decide which way out to send it, and to whom.
6. Step five: it subtracts one from the hop counter. If the counter hits zero, the packet is destroyed and a complaint is sent back.
7. Step six: because it changed the hop counter, it fixes the small checksum.
8. Step seven: it finds the hardware address of the next machine along.

9. Step eight: it builds a brand new outer wrapper with that hardware address and pushes the packet out of the chosen wire.
10. The single most important thing to hold on to: the outer wrapper is destroyed and rebuilt at every hop, but the address inside stays the same.

26.5.2 PLAIN – a picture in your head

1. Imagine a parcel travelling by a chain of couriers, each covering one leg.
2. The parcel has the final address written on it in permanent ink. Nobody ever changes that.
3. Each courier puts the parcel in their own bag with their own routing sticker: "from depot 4, to depot 9".
4. When it arrives at depot 9, that bag is thrown in the bin. The sticker was only ever about that one leg.
5. Depot 9 reads the permanent address, decides depot 14 is next, puts the parcel in a fresh bag with a fresh sticker, and sends it on.
6. Where this comparison breaks: real couriers know the whole route. A router knows only the next depot. It has no idea how many more legs remain.

26.5.3 PLAIN – a worked example

1. Follow the reader's first packet across three real hops from the traceroute.

```
laptop           192.168.0.1        172.31.0.17     137.97.29.249
   |                   |                  |                |
   |== frame A ======>|                  |                |
   |   MAC: me->rtr   |== frame B =======>|                |
   |                   |   MAC: rtr->isp   |== frame C ====>|
   |                   |                  |  MAC: isp->hop3|
   |                   |                  |                |
 TTL 64            TTL 63              TTL 62          TTL 61
```

2. Three different frames carry one packet. Each frame lives for exactly one link and is then discarded.
3. The MAC addresses below are illustrative, since the session notes do not record them, but the pattern is exact.

Frame	Source MAC	Destination MAC	TTL
A	laptop card	router LAN port	64
B	router WAN port	ISP access port	63
C	ISP access port	next ISP port	62

4. Now the same journey seen from the IP header.

Field	Frame A	Frame B	Frame C
Dest IP	20.207.73.82	unchanged	unchanged
Source IP	192.168.0.14	router WAN	unchanged
TTL	64	63	62
Checksum	d1a5	d2a5	d3a5

5. Notice the checksum column. Each hop adds `0x0100`, because the TTL is the high byte of a 16 bit word and it went down by one.
6. The honest version: the source IP address is not really unchanged. The reader's home router does network address translation, so it rewrites the source from `192.168.0.14` to its own public-facing address.

26.5.4 PLAIN - what is really happening inside

1. Step seven above, finding the hardware address of the next machine, deserves its own explanation, because it is where two different address systems meet.
2. The routing table gives the router an IP address for the next hop, not a hardware address.
3. To build the new frame it needs a hardware address. It gets one by asking on the local wire: "who has this IP address, tell me your hardware address".
4. That question and answer is the Address Resolution Protocol. The answers are kept in a small table so the question is only asked occasionally.
5. If the next hop is not a router at all but the final destination on the same wire, the same lookup happens for the destination address itself.
6. Now the speed problem. Doing a full table lookup in software for every packet is far too slow for a serious router.
7. So real routers split in two. The **control plane** works out what the routes should be, slowly, in software.
8. The **data plane** forwards packets, fast, using a stripped-down copy of the answers built into dedicated hardware.

26.5.5 TECHNICAL - the engineer's version

1. The full forwarding path for an IPv4 unicast packet, per RFC 1812:

```
1) receive frame, verify FCS, check dst MAC is ours
2) demultiplex on EtherType 0x0800 -> IPv4
3) verify IHL, total length and header checksum
4) if TTL <= 1: drop, send ICMP time exceeded
5) longest prefix match on destination in the FIB
6) decrement TTL, update header checksum incrementally
7) resolve next-hop IP to MAC via ARP cache or request
8) encapsulate in a new L2 header, enqueue on egress
```

2. Those eight steps are the forwarding plane. Nothing in the list consults any state about previous packets, which is what makes IP stateless.
3. The routing table is the Routing Information Base, the RIB. The hardware copy is the Forwarding Information Base, the FIB.
4. The RIB holds every route learned from every source, including losers. The FIB holds only the winner for each prefix, in a form the silicon can search.
5. FIB lookup in hardware is typically a TCAM, ternary content addressable memory, which compares a key against every stored prefix in one clock cycle.
6. TCAM is expensive and power hungry, which is why routers have hard limits on route counts. A switch sold as supporting 16,000 IPv4 routes cannot hold a full internet table.
7. As of the CIDR Report for 13 August 2026, the global IPv4 table held about 1,071,273 prefixes originated by 79,217 autonomous systems.
8. Address resolution is ARP for IPv4, RFC 826 from November 1982. IPv6 uses Neighbor Discovery instead, RFC 4861.
9. Tools: `arp -a` on macOS lists the resolved neighbours; `netstat -rn` shows the routing table; `ip route get 20.207.73.82` on Linux shows the exact decision for one destination.

26.5.6 WORDS - remember these

1. Hop — one link between two routers — a single layer 3 forwarding step, which decrements TTL by one.
2. Next hop — who to hand the packet to next — the IP address of the adjacent router selected by the routing lookup.
3. Control plane — the part that decides routes — routing protocol software building the RIB.
4. Data plane — the part that moves packets — hardware forwarding using the FIB.

5. RIB — the full list of everything learned — Routing Information Base, holding all candidate routes.
6. FIB — the short list actually used — Forwarding Information Base, one best next hop per prefix, held in fast memory.

26.6 The routing table

26.6.1 PLAIN - in simple words

1. A routing table is a list of rules of the form "packets going here should be sent that way".
2. Each rule has a destination pattern, a way out, and sometimes a next machine to hand the packet to.
3. The destination pattern is not one address. It is a whole block of addresses, written as an address and a slash number.
4. `192.168.0.0/24` means "any address whose first 24 bits match", which is the 256 addresses from `192.168.0.0` to `192.168.0.255`.
5. Several rules can match the same address at once. That is normal and deliberate.
6. When several match, the winner is the one with the biggest slash number, the most specific one. That rule is called **longest prefix match**.
7. There is almost always one rule with a slash number of zero, written `0.0.0.0/0`. Zero bits must match, so it matches everything.
8. That is the **default route**, meaning "if nothing else matched, send it here". On the reader's laptop it points at `192.168.0.1`.

26.6.2 PLAIN - a picture in your head

1. Think of the signs in a large airport.
2. One sign says "All departures, this way". That is the default route.
3. Further on, a sign says "International departures, this way". More specific, so it overrides the general one.
4. Further on again, "Flights to Mumbai, gate 34". More specific still.
5. All four signs point at your flight. You obey the most specific one that applies to you, and you ignore the vaguer ones.
6. Where this comparison breaks: an airport arranges the signs so you meet the general one first and the specific one later. A router sees all its rules at once and picks the most specific in a single step.

26.6.3 PLAIN - a worked example

1. Take a router with these five rules, and a packet for `20.207.73.82`, which is the address the reader's `github.com` resolved to.

Rule	Next hop	Matches?
0.0.0.0/0	192.168.0.1	yes
20.0.0.0/8	10.9.0.1	yes
20.192.0.0/10	10.9.0.5	yes
20.207.64.0/18	10.9.0.9	yes
20.207.73.0/24	10.9.0.13	yes

2. All five match. Now check the bits and see why.

```
20.207.73.82  = 00010100 11001111 01001001 01010010

/0    (nothing to compare)           match
/8    00010100                       match
/10   00010100 11                    match
```

```
/18   00010100 11001111 01            match
/24   00010100 11001111 01001001      match
```

3. Take the third one slowly. `20.192.0.0/10` compares the first 10 bits.
4. The route's second byte is 192, which is `11000000`. Its first two bits are `11`.
5. The address's second byte is 207, which is `11001111`. Its first two bits are also `11`. So the first 10 bits agree.
6. Now a rule that does not match, for contrast: `20.207.128.0/18`. The third byte 128 is `10000000`, whose first two bits are `10`, and the address's third byte 73 is `01001001`, whose first two bits are `01`. No match.
7. Winner: `20.207.73.0/24`, because 24 is the largest number of matching bits. The packet goes to `10.9.0.13`.
8. The default route loses every time anything else matches, which is exactly what "default" means.

26.6.4 PLAIN – what is really happening inside

1. Routes get into the table in four different ways, and the difference matters.
2. **Connected routes** appear automatically. Give an interface the address `192.168.0.14/24` and the machine knows `192.168.0.0/24` is reachable directly on that interface, with no next hop at all.
3. **Static routes** are typed in by a person or set by a script. They stay until someone removes them.
4. **Dynamic routes** are learned from a routing protocol talking to other routers. Section 26.7 covers those.
5. When two sources offer a route to the very same prefix, longest prefix match cannot break the tie, because the prefixes are identical.
6. Two more tie-breakers exist, in this order.
7. First, **administrative distance**: a trust score per source. A lower number means "trust this source more". Connected beats static beats OSPF beats RIP.
8. Second, **metric**: within one protocol, the cheaper route wins. What "cheaper" means depends entirely on the protocol.

26.6.5 TECHNICAL – the engineer's version

1. Here is a realistic `netstat -rn` from the reader's macOS machine. Only the IPv4 section is shown, because the machine reported `IPv6: (none)`.

```
Routing tables

Internet:
Destination        Gateway            Flags      Netif Expire
default            192.168.0.1        UGScg        en0
127                127.0.0.1          UCS          lo0
127.0.0.1          127.0.0.1          UH           lo0
169.254            link#12            UCS          en0      !
192.168.0          link#12            UCS          en0      !
192.168.0.1/32     link#12            UCS          en0      !
192.168.0.1        ac:22:0b:4e:1f:2a  UHLWIir      en0    978
192.168.0.14/32    link#12            UCS          en0      !
192.168.0.255      ff:ff:ff:ff:ff:ff  UHLWbI       en0      !
224.0.0/4          link#12            UmCS         en0      !
255.255.255.255/32 link#12            UCS          en0      !
```

2. Line by line:
   1. `default 192.168.0.1` is `0.0.0.0/0` via the home router. Every packet for `20.207.73.82` used this line.
   2. `127` and `127.0.0.1` are loopback, so the machine can talk to itself without touching any wire.
   3. `169.254` is the link-local block from RFC 3927, used when DHCP fails.
   4. `192.168.0` is the connected route for the home network, reachable directly on `en0` with no gateway.

5. `192.168.0.1/32` and `192.168.0.14/32` are host routes macOS creates for the gateway and for the machine's own address.
6. `192.168.0.1 ac:22:0b:4e:1f:2a` is the resolved ARP entry for the router, with 978 seconds left before it expires.
7. `192.168.0.255` is the broadcast address for the subnet.
8. `224.0.0/4` is the multicast block.
9. `255.255.255.255/32` is the all-hosts broadcast address.

3. The Flags column is a set of letters, documented in the macOS `netstat` manual page.

Flag	Name	Meaning
U	RTF_UP	Route is usable
G	RTF_GATEWAY	Send via an intermediary
S	RTF_STATIC	Added manually
H	RTF_HOST	A single host, not a net
L	RTF_LLINFO	Has link-address info
W	RTF_WASCLONED	Generated by cloning
I	RTF_IFSCOPE	Scoped to one interface
i	RTF_IFREF	Holds an interface reference
r	RTF_ROUTER	Peer is a default router
C	RTF_CLONING	Spawns routes when used
c	RTF_PRCLONING	Protocol-specified cloning
m	RTF_MULTICAST	A multicast address
b	RTF_BROADCAST	A broadcast address
g	RTF_GLOBAL	Destination is on the global internet

4. So `UGScg` on the default route reads: usable, via a gateway, static, protocol-cloning, global. The `!` on some lines marks a reject route for addresses inside the block with no more specific entry.
5. Administrative distance is a Cisco convention, not an internet standard. Other vendors use the term route preference with different numbers. Juniper's defaults, for instance, differ from the list below.

Source	Cisco distance	Notes
Connected	0	Trusted absolutely
Static	1	Typed by a human
eBGP	20	From another AS
EIGRP internal	90	Cisco only
OSPF	110	Link state
IS-IS	115	Link state
RIP	120	Distance vector
iBGP	200	Inside one AS

6. Tools: `netstat -rn` on macOS and BSD, `ip route show` on Linux, `route print` on Windows, `show ip route` on Cisco IOS, and `route get 20.207.73.82` on macOS to see exactly which line wins.

26.6.6 WORDS - remember these

1. Prefix — a block of addresses written with a slash — an address plus a mask length, such as `20.192.0.0/10`.
2. Longest prefix match — the most specific sign wins — selecting the matching route with the greatest mask length.

3. Default route — where to send anything unrecognized — `0.0.0.0/0`, matching every destination with zero bits of agreement required.
4. Connected route — the network you are plugged into — a route derived automatically from an interface address and mask.
5. Static route — a line typed by hand — an administratively configured route with no protocol maintaining it.

26.7 How routers learn routes

26.7.1 PLAIN - in simple words

1. A small network can have its routes typed in by hand. A large one cannot.
2. So routers talk to each other and tell each other what they can reach.
3. There are two families of ways to do this, and they think very differently.
4. The first family is **distance vector**. Each router tells its neighbours a list: "I can reach these places, and here is how far away each one is".
5. A router receiving that list adds the cost of the link it came over and compares the result with what it already knew.
6. The second family is **link state**. Every router describes only its own direct connections, and that description is flooded to every other router.
7. Every router therefore ends up with an identical picture of the whole network, and computes the best paths itself.
8. Distance vector is "ask a neighbour for directions". Link state is "everyone gets the same map and works it out".

26.7.2 PLAIN - a picture in your head

1. Imagine a village with no maps, where you find your way by asking people.
2. Distance vector is asking your neighbour, "how far is the market?" They say "twelve minutes that way". You add the two minutes to reach them and tell the next person "fourteen minutes".
3. This works, but nobody can check anything. If your neighbour is wrong, everyone downstream of you is wrong too, and confidently so.
4. Worse: if the market closes, your neighbour hears from you that the market is fourteen minutes away, and tells you it is sixteen, and you tell them eighteen. The number climbs forever while the market stays shut.
5. Link state is different. Every person writes down only what they can see from their own doorstep, and copies of every note are posted to everybody.
6. Now everyone holds the same stack of notes, draws the same map, and works out their own shortest path.
7. Where this comparison breaks: real people would not re-post the entire note stack every thirty minutes, but link state protocols do refresh, and the flooding costs real bandwidth on very large networks.

26.7.3 PLAIN - a worked example

1. Four routers, A, B, C and D, in a line, with A connected to a network N.

```
N --- A --- B --- C --- D
```

2. With RIP, which counts hops, the table for network N fills in over time. A says N = 0 throughout, because N is connected to it.

Round	B says	C says	D says
0	unknown	unknown	unknown

Round	B says	C says	D says
1	N = 1	unknown	unknown
2	N = 1	N = 2	unknown
3	N = 1	N = 2	N = 3

3. Each round is one update interval. RIP's default update interval is 30 seconds, so full knowledge of a four-router line takes about a minute and a half.
4. Now break the link between A and B. A knows immediately. B does not.
5. Without protection, B hears from C that N is reachable at distance 2, adds one, and announces N at distance 3. C then hears 3 and announces 4.
6. The count climbs: 3, 4, 5, and onward, one per update round, until it reaches 16, which RIP defines as unreachable. That takes many rounds.
7. This is the famous **count to infinity** problem, and it is why RIP declares 16 to mean unreachable rather than allowing large numbers.

26.7.4 PLAIN - what is really happening inside

1. Link state protocols need a way to turn a map into best paths, and they all use the same method: **Dijkstra's algorithm**.
2. Here it is in plain words, with no mathematics.
3. Write down every router. Give yourself a cost of 0 and everyone else a cost of infinity. Mark nobody as finished.
4. Pick the unfinished router with the smallest cost. The first time round that is you.
5. Look at each of its neighbours. For each one, work out the cost of reaching it through the router you just picked.
6. If that total beats the neighbour's current cost, write down the better total and remember which router you came from.
7. Mark the picked router finished. It can never improve, because every other route to it would have to go through something more expensive.
8. Repeat: pick the smallest unfinished one, relax its neighbours, mark it finished.
9. Stop when everyone is finished. Following the "came from" notes backwards gives the shortest path to every destination.
10. The output is a tree rooted at you, called the shortest path tree, and the first step of each branch is the next hop you put in the routing table.
11. **Convergence** is the time from a change happening to every router having finished redoing this and agreeing again.

26.7.5 TECHNICAL - the engineer's version

1. Interior gateway protocols run inside one administrative domain. Exterior gateway protocols run between domains. Sections 26.8 and 26.9 cover the exterior side.

Protocol	Family	Metric	Specification
RIPv1	Distance vector	Hop count	RFC 1058, 1988
RIPv2	Distance vector	Hop count	RFC 2453, 1998
OSPFv2	Link state	Cost	RFC 2328, 1998
IS-IS	Link state	Wide metric	ISO 10589, RFC 1195
EIGRP	Advanced DV	Composite	RFC 7868, 2016

2. RIP runs over UDP port 520, updates every 30 seconds, times a route out after 180 seconds, and treats metric 16 as infinity. RIPv2 added subnet masks and uses multicast address `224.0.0.9`.

3. OSPFv2 was specified by John Moy. Version 1 appeared as RFC 1131 in 1989 and the current version 2 as RFC 2328 in April 1998. OSPFv3 for IPv6 is RFC 5340.
4. OSPF runs directly over IP as protocol number 89. On a broadcast network it sends hellos every 10 seconds and declares a neighbour dead after 40 seconds.
5. OSPF divides a large network into areas, all joined to area 0, the backbone. Link state flooding is confined within an area, which bounds both the database size and the computation.
6. IS-IS comes from the OSI world, standardized as ISO/IEC 10589, and was adapted to carry IP by RFC 1195 in December 1990. Radia Perlman designed it at Digital Equipment Corporation for DECnet Phase V.
7. IS-IS runs directly on the link layer rather than inside IP, which is one reason very large carriers favour it: it is harder to attack from an IP packet, and it carries IPv4 and IPv6 in one protocol without change.
8. Dijkstra's algorithm was conceived by Edsger Dijkstra in 1956 and published in Numerische Mathematik in 1959. Its cost with a binary heap is on the order of E log V, where E is links and V is routers.

Convergence stage	Typical time
Link failure detection	10 ms to 40 s
LSA flooding	10 to 100 ms
SPF computation	1 to 100 ms
FIB update	10 ms to seconds

9. Tools: `show ip ospf neighbor` and `show isis adjacency` on router command lines, and `birdc show route` or `vtysh` for the open source stacks BIRD and FRRouting.

26.7.6 WORDS - remember these

1. Distance vector — ask a neighbour how far it is — a protocol where routers exchange destination and metric pairs, without topology knowledge.
2. Link state — everyone gets the same map — a protocol where routers flood descriptions of their own links and each computes paths independently.
3. Count to infinity — a distance that climbs while the destination is gone — a distance vector loop where metrics increment until reaching the infinity value.
4. Split horizon — do not tell a neighbour what it told you — suppressing advertisement of a route out of the interface it was learned on.
5. Dijkstra's algorithm — repeatedly finish the cheapest unfinished node — a shortest path first algorithm, published 1959, used by OSPF and IS-IS.
6. Convergence — everyone agreeing again after a change — the interval from a topology change to a consistent forwarding state network-wide.

26.8 Autonomous systems

26.8.1 PLAIN - in simple words

1. The internet is not one network. It is about eighty thousand separate networks that have agreed to carry each other's traffic.
2. Each of those separate networks is called an **autonomous system**, usually shortened to AS.
3. Autonomous means it decides its own internal routing for itself. Nobody outside gets a say in how it moves packets inside its own borders.
4. Each AS has a number. Microsoft's main one is AS8075. That number is how networks refer to each other.
5. The reader's packets started in the ISP's autonomous system and ended up in Microsoft's, which is why the names at hop 7 change to `ntwk.msn.net`.

6. Networks connect to each other in two very different commercial ways.
7. **Transit** is buying. You pay a bigger network to carry your traffic to the whole internet and to bring the whole internet back to you.
8. **Peering** is swapping. Two networks connect directly and exchange traffic between their own customers, usually with no money changing hands.

26.8.2 PLAIN - a picture in your head

1. Think of the world's postal services.
2. Each country runs its own post inside its borders however it likes. Different vans, different sorting rules, different rates. That is autonomy.
3. For letters leaving the country, two arrangements exist.
4. A small country might pay a big neighbour to handle everything international. That is transit. One bill, global reach.
5. Two countries that exchange a lot of post might set up a direct exchange at the border and simply swap sacks, no invoice. That is peering.
6. A border exchange building where twenty countries all bring sacks and swap with whoever they like is an internet exchange point.
7. Where this comparison breaks: postal countries have fixed geography and cannot choose their neighbours. Networks connect to whoever they wish, anywhere, so an AS in India can peer directly with one in Germany.

26.8.3 PLAIN - a worked example

1. Take the address at the end of the reader's story, `20.207.73.82`.
2. Any of these three commands tells you which AS owns it.

```
whois -h whois.cymru.com " -v 20.207.73.82"
dig +short 82.73.207.20.origin.asn.cymru.com TXT
whois 20.207.73.82
```

3. The second one reverses the four octets and asks a special DNS zone. It is the fastest, because it is just a DNS lookup.
4. The answer for this address, checked in August 2026: AS8075, Microsoft Corporation, with the covering route `20.192.0.0/10`, registered in the United States, and the address itself geolocating to Pune, India.
5. That last detail lines up neatly with the trace. Hops 10, 11 and 12 carry `pnq20` and `pnq21` in their names, and `pnq` is the airport code for Pune.
6. AS8075 is registered under the name MICROSOFT-CORP-MSN-AS-BLOCK, which is also why the hop names use `msn.net`.

26.8.4 PLAIN - what is really happening inside

1. Networks are informally sorted into tiers, and the definition is about whether you pay anyone.
2. A **tier 1** network can reach every destination on the internet without buying transit from anybody. It peers with all the other tier 1 networks.
3. A **tier 2** network peers with many networks but still buys transit from at least one provider to reach everything.
4. A **tier 3** network buys transit and does little or no peering. Most small ISPs are here.
5. Experts disagree about who counts as tier 1, because the list depends on private contracts nobody publishes. Treat any published list as an informed estimate.
6. An **internet exchange point**, or IXP, is a neutral room with a large ethernet switch in it.
7. Every member plugs one port into that switch. Once connected, a member can exchange traffic with any other member that agrees to peer.
8. The exchange does not route anything. It provides a shared layer 2 fabric. The peering itself is BGP sessions between members over that fabric.

9. Most exchanges also run a **route server**, so a new member can get routes from dozens of others by setting up one session instead of dozens.

26.8.5 TECHNICAL - the engineer's version

1. AS numbers were originally 16 bit, giving 0 to 65,535. RFC 6793, December 2012, extended them to 32 bit, giving 0 to 4,294,967,295.

Range	Purpose	Reference
0	Reserved	RFC 7607
1 to 64495	Public, 16 bit	IANA to RIRs
64496 to 64511	Documentation	RFC 5398
64512 to 65534	Private use	RFC 6996
65535	Reserved	IANA
65536 to 65551	Documentation	RFC 5398
65552 to 4199999999	Public, 32 bit	IANA to RIRs
4200000000 to 4294967294	Private use	RFC 6996

2. AS23456 is reserved as AS_TRANS, a placeholder used when a 32 bit AS number must be shown to a router that only understands 16 bit numbers.
3. IANA allocates blocks of AS numbers to the five regional internet registries, ARIN, RIPE NCC, APNIC, LACNIC and AFRINIC, which then assign them.
4. Real AS numbers worth knowing:

AS number	Network	Role
8075	Microsoft	Owns 20.207.73.82
13335	Cloudflare	Runs the 1.1.1.1 resolver
15169	Google	Content and cloud
16509	Amazon	Most IPv4 prefixes: 15,814
32934	Meta, formerly Facebook	Subject of 26.9
3356	Lumen, formerly Level 3	Large transit
174	Cogent	Large transit
1299	Arelion, formerly Telia	Large transit
2914	NTT	Large transit
6453	Tata Communications	Large transit, India
3491	PCCW Global	Large transit
17557	Pakistan Telecom	Subject of 26.9

5. The reader's resolver, `1.1.1.1`, is announced by AS13335. The name the resolver returned, `20.207.73.82`, sits inside AS8075. Two entirely separate networks were involved before a single packet went anywhere.
6. Real internet exchange figures, all checkable:

Exchange	Figure	Date
DE-CIX, all sites	25 Tbit/s peak	8 April 2025
DE-CIX Frankfurt	Launched 1995	1995
DE-CIX Frankfurt	About 1,100 networks	April 2025
AMS-IX Amsterdam	15 Tbit/s peak	15 April 2026
India, all IXPs	31 active, 934 members	August 2026

Exchange	Figure	Date

7. Tools: `whois -h whois.cymru.com`, the PeeringDB directory for who peers where, `bgp.he.net` and `bgp.tools` for AS relationships, and the CIDR Report for daily table statistics.

26.8.6 WORDS - remember these

1. Autonomous system — one network under one routing policy — a set of prefixes with a single clearly defined routing policy, identified by an AS number.
2. ASN — the number that names a network — a 16 or 32 bit autonomous system number allocated by IANA through a regional registry.
3. Transit — paying someone for global reach — a paid service where a provider announces your prefixes to the whole internet and carries your traffic.
4. Peering — swapping traffic directly — an interconnection carrying only the two parties' own customer routes.
5. Settlement-free — peering with no invoice — peering where neither side pays the other, the normal arrangement between similar-sized networks.
6. IXP — a shared room where networks plug in and swap — an internet exchange point providing a neutral layer 2 fabric for bilateral BGP peering.

26.9 BGP, the protocol that glues the internet together

26.9.1 PLAIN - in simple words

1. Inside one network, routers use protocols like OSPF to find the best path.
2. Between networks, none of that works, because the two sides do not trust each other, do not share a map, and have contracts to honour.
3. So there is a second protocol just for the boundaries between autonomous systems. It is called the **Border Gateway Protocol**, or BGP.
4. BGP is how one network announces to another: "I can reach these address blocks, and here is the list of networks a packet would cross to get there".
5. That list of networks is called the **AS path**, and it is the heart of the protocol.
6. There are only two things BGP ever says. "I can reach this block", which is an announcement, and "I can no longer reach this block", which is a withdrawal.
7. Here is the part that surprises engineers: BGP does not pick the fastest path. It picks the path its operator's policy prefers.
8. Policy usually means money. A route through a network you do not pay is preferred over a route through a network you do pay, even if it is longer.
9. And here is the uncomfortable part: in the original design, BGP has no way to check whether an announcement is true.

26.9.2 PLAIN - a picture in your head

1. Imagine a shipping company deciding how to send goods to Chennai.
2. Three freight brokers phone in. Each says "I can get it there", and each gives you a chain of firms it would pass through.
3. Broker A quotes a chain of two firms. Broker B quotes three. Broker C quotes two, but one of them charges you per container while the others do not.
4. You do not pick the shortest chain. You pick the one that costs you least, then use chain length only to break ties.
5. Now the flaw. Nobody verifies that any broker can actually reach Chennai. Whoever claims the most direct-sounding route gets the goods.

6. Where this comparison breaks: a shipping company would notice quickly that its goods never arrived. On the internet, the sender often sees only silence, which is exactly the shape of the reader's failed connection.

26.9.3 PLAIN – a worked example

1. This is the YouTube hijack of Sunday 24 February 2008, one of the clearest real examples ever recorded, documented by the RIPE NCC.
2. YouTube, then AS36561, announced `208.65.152.0/22`, a block containing its servers.
3. Pakistan's government ordered access to YouTube blocked. Pakistan Telecom, AS17557, decided to do it by announcing a more specific route internally so its own users would be sent nowhere.
4. At 18:47 UTC, AS17557 began announcing `208.65.153.0/24`. That is a /24 inside YouTube's /22, so it is more specific.
5. The announcement escaped. Pakistan Telecom's upstream provider, PCCW Global, AS3491, accepted it and passed it to the rest of the world.
6. Longest prefix match then did what it always does. A /24 beats a /22, so routers everywhere sent YouTube traffic to Pakistan Telecom.

Time UTC	Event
18:47	AS17557 announces 208.65.153.0/24
20:07	AS36561 announces the same /24 back
20:51	AS17557 path seen prepended
21:01	AS3491 withdraws AS17557 prefixes

7. Notice YouTube's counter-move at 20:07. It announced the same /24 itself, so the two announcements were equally specific and the tie fell to AS path length. That recovered some traffic but not all.
8. It ended only when PCCW stopped accepting the announcement at 21:01.

26.9.4 PLAIN – what is really happening inside

1. BGP is a **path vector** protocol. Each announcement carries the full list of autonomous systems it has crossed, not just a distance.
2. That list has two jobs. It breaks loops, because a router discards any announcement that already contains its own AS number. And it is a tie-break, because a shorter list is preferred.
3. Two kinds of BGP session exist and they behave differently.
4. **eBGP** runs between two different autonomous systems. Every time an announcement crosses an eBGP session, the sender adds its own AS number to the front of the path.
5. **iBGP** runs between routers inside one autonomous system. The AS path is not changed, because the announcement has not left the AS.
6. iBGP has a rule that catches everyone out: a router will not pass a route learned from one iBGP peer on to another iBGP peer. That prevents loops, but it means every iBGP router must talk to every other one, or you need route reflectors.
7. When several announcements offer the same prefix, BGP runs a fixed list of tie-breakers, in strict order, and stops at the first one that decides.
8. The first real tie-break is **local preference**, a number set by your own policy. It exists to encode "prefer the free path over the paid path".
9. Only after local preference does AS path length matter. This is why the internet's routes are frequently not the shortest ones available.

26.9.5 TECHNICAL - the engineer's version

1. BGP-4 is specified in RFC 4271, January 2006, which obsoleted RFC 1771 from
1995. The first version, RFC 1105, dates from June 1989 and was written by Kirk Lougheed of Cisco and Yakov Rekhter of IBM.
2. The best path selection order, as implemented by Cisco IOS. Steps 1 and parts of the ordering are vendor conventions, not parts of RFC 4271.

Step	Criterion	Prefer
1	Weight	Highest, Cisco only, local
2	Local preference	Highest
3	Locally originated	Self-originated
4	AS path length	Shortest
5	Origin type	IGP over EGP over incomplete
6	MED	Lowest
7	eBGP versus iBGP	eBGP
8	IGP metric to next hop	Lowest
9	Router ID	Lowest

3. Read that table as a statement about economics. The two strongest knobs, weight and local preference, are both purely local policy. Distance does not appear until step 4, and physical latency never appears at all.
4. Three incidents worth knowing exactly:
    1. 25 April 1997, the AS 7007 incident. A router at a small Florida provider re-announced large parts of the internet as /24s, and much of the global routing system followed the more specific routes into it.
    2. 24 February 2008, the Pakistan Telecom YouTube hijack described above.
    3. 4 October 2021, the Facebook outage. Cloudflare recorded a burst of BGP updates from Facebook's network beginning around 15:39 UTC, including withdrawal of the routes covering all of its authoritative DNS servers.
5. The Facebook case is the cleanest illustration of withdrawal. Facebook's DNS servers were designed to withdraw their own routes if they lost contact with the data centres, and a backbone configuration change triggered exactly that.
6. Resolvers worldwide, including the `1.1.1.1` service the reader uses, then returned SERVFAIL for `facebook.com`, because the name servers were not merely down, they were unreachable at the routing layer.
7. BGP activity resumed shortly before 21:00 UTC, routing was restored at about 21:50 UTC, and services were generally usable again by about 22:45 UTC. Six to seven hours, from a routing change, not an attack.
8. A **route leak** is different from a hijack. The prefixes are genuine, but they are passed to a party that should never have received them, usually by a network announcing its providers' routes to its other providers.
9. On 16 April 2021, AS55410, Vodafone Idea in India, leaked more than 30,000 prefixes it did not own, drawing a large multiple of its normal inbound traffic and disrupting reachability well beyond India.
10. The current defence is the **Resource Public Key Infrastructure**, RPKI, architecture in RFC 6480, February 2012.
11. An address holder publishes a signed **Route Origin Authorization**, a ROA, saying "AS number X is authorized to originate this prefix, up to this length". `20.207.73.82` is covered by a valid ROA today.
12. Routers then perform **Route Origin Validation**, RFC 6811, January 2013, marking each route Valid, Invalid or NotFound, and operators normally drop the Invalid ones.
13. Deployment figures from the NIST RPKI Monitor for 1 August 2026, IPv4:

State	Prefix-origin pairs	Share
Valid	842,771	68.57 percent
NotFound	372,904	30.34 percent
Invalid	13,370	1.09 percent

14. Measured by address space rather than by prefix count, 61.64 percent of announced /24 equivalents were covered by a valid ROA on the same date.
15. Coverage is not the same as enforcement. Published assessments in 2026 put the share of autonomous systems actually enforcing ROV at roughly a quarter, and this is the number that matters.
16. RPKI origin validation only proves who may originate a prefix. It says nothing about whether the rest of the AS path is real, so it stops the 2008 style hijack but not every route leak.
17. Tools: `birdc show protocols`, RIPE RIS and RouteViews for historical BGP data, BGPlay for animated replays of incidents, and public looking glasses for a live view from another network.

26.9.6 WORDS - remember these

1. BGP — how networks tell each other what they can reach — the Border Gateway Protocol, RFC 4271, running over TCP port 179.
2. Path vector — an announcement carrying its full route history — a protocol where each update lists the AS numbers traversed.
3. AS path — the list of networks a route crossed — the AS_PATH attribute, used for loop detection and as a tie-break.
4. Announcement — "I can reach this block" — a BGP UPDATE advertising NLRI with path attributes.
5. Withdrawal — "I can no longer reach this block" — a BGP UPDATE removing a previously advertised prefix.
6. Local preference — a policy dial for choosing paid versus free — the LOCAL_PREF attribute, compared before AS path length.
7. eBGP — a session between two networks — external BGP, which prepends the local AS number to the path.
8. iBGP — a session inside one network — internal BGP, which does not change the AS path and does not re-advertise between iBGP peers.
9. Hijack — announcing address space that is not yours — an unauthorized origin announcement, valid to every router that hears it.
10. Route leak — passing routes to somebody who should not get them — propagating routes in violation of the intended business relationship.

26.10 Why nobody knows the whole path in advance

26.10.1 PLAIN - in simple words

1. There is no plan for a packet's journey. Nowhere is one written down.
2. Each router knows only one thing about a destination: which neighbour to hand the packet to next.
3. It does not know how many hops remain, or which countries they are in, or whether the far end is even up.
4. This is called **hop-by-hop forwarding**, and it is the whole model.
5. The path exists only as the sum of many independent local decisions, made fresh for every packet.
6. So the path can change in the middle of a conversation, without warning, and nothing has to be renegotiated.
7. And the way back need not use the same routers as the way out. The two directions are decided by different networks with different policies.

26.10.2 PLAIN - a picture in your head

1. Imagine posting a letter and asking every person you meet, "which way to Pune?", then walking one street in the direction they point.
2. At the next corner you ask again. Nobody you meet knows the whole route. Each knows only the next turn.
3. You will get there, because each person is right about their own street.
4. If a street closes while you are walking, the next person simply points a different way. Nothing had to be cancelled.
5. Where this comparison breaks: you can see where you have been. A packet cannot. It carries no record of its journey unless a rare option is enabled.

26.10.3 PLAIN - a worked example

1. The reader's traceroute shows 12 hops outward before the replies stop.
2. Hop 1 is the router in the flat. Hops 2 to 6 are private ISP addresses. Hop 3 is the ISP's public address `137.97.29.249`. From hop 7 the names become `ntwk.msn.net`, so the packets are inside Microsoft's network.
3. The names then read Delhi, then Mumbai, then Pune. That is a real geographic journey, visible in the site codes `del01`, `bom01`, `pnq20` and `pnq21`.
4. Now the honest limit. Every one of those 12 lines was learned from a reply sent back to the reader.
5. A reply is generated when a packet's TTL hits zero. The reply is a brand new packet, sent from that router to the reader, over whatever path that router chooses.
6. So each line tells you one address on the forward path, and nothing whatever about how the answer travelled home.
7. If the fault lies on the return path, a traceroute from the reader's laptop cannot see it at all. That is a limit of the model, not of the tool.

26.10.4 PLAIN - what is really happening inside

1. Paths differing in the two directions is called **asymmetric routing**, and it is the normal case on the internet, not the exception.
2. It happens because forward and reverse decisions are made by different parties. Your ISP decides how to reach Microsoft. Microsoft decides how to reach your ISP.
3. Both apply their own local preference, so both may pick a cheaper path than the other side chose.
4. A stateful firewall that only sees one direction of a flow will drop it, because it never saw the handshake it expects.
5. To diagnose properly you need a trace from both ends, or a looking glass at the far network that can trace back towards you.

26.10.5 TECHNICAL - the engineer's version

1. Traceroute was written by Van Jacobson in 1987 and released in 1988. It works by sending probes with TTL 1, 2, 3 and upward, and reading the ICMP time exceeded messages, ICMP type 11 code 0, from RFC 792.
2. Classic Unix traceroute uses UDP to high destination ports. macOS supports `-I` for ICMP echo probes and `-T` for TCP probes.
3. `traceroute -T -p 443 20.207.73.82` is the version worth running for the reader's fault, because it probes the exact port that failed. Firewalls frequently treat UDP, ICMP and TCP port 443 completely differently.
4. Silence at the end of a trace proves nothing on its own. Routers commonly rate limit ICMP generation, and many networks suppress it entirely. Hops 13 to 20 showing `* * *` is consistent with a healthy path.
5. Reverse path visibility requires cooperation: a public looking glass, a RIPE Atlas measurement from a probe near the target, or `mtr` run from both ends.

6. `mtr` is the better everyday tool, because it runs continuously and shows per-hop loss and jitter rather than a single snapshot.

26.10.6 WORDS - remember these

1. Hop-by-hop forwarding — each router only knows the next step — destination based forwarding with no end-to-end path state.
2. Asymmetric routing — the way back is not the way out — forward and reverse paths differing because each direction is chosen by a different AS.
3. Time exceeded — the message a router sends when the counter runs out — ICMP type 11 code 0, the mechanism traceroute depends on.
4. Looking glass — a window into another network's view — a public interface for running traceroute or BGP queries from a remote AS.
5. ICMP rate limiting — a router answering only some probes — deliberate throttling of ICMP generation, which shows as false loss in traceroute.

26.11 Load balancing inside the network

26.11.1 PLAIN - in simple words

1. Between two big routers there is rarely one cable. There are often four, eight or sixteen, run in parallel for capacity and safety.
2. When a router has several equally good ways to reach a destination, it uses all of them. This is called **equal cost multipath**, or ECMP.
3. Now the question: how does it decide which cable each packet takes?
4. The safe method is to keep each conversation on one cable. All packets of one connection follow the same link, in order.
5. The router works out which cable by mixing a few fields from the packet header into a number and taking the remainder. The same conversation always produces the same number.
6. This is why the reader's traceroute shows two or three different addresses at hops 5, 6, 8, 9 and 11.
7. Traceroute sends many probe packets, and probes are usually not all part of one conversation, so they scatter across the parallel links and come back with different router addresses.
8. That is not a fault. It is the visible fingerprint of a healthy, redundant network, and it is PROVEN evidence of parallel paths rather than of trouble.

26.11.2 PLAIN - a picture in your head

1. Think of a toll plaza with eight lanes going the same way.
2. If you sent every car down whichever lane was shortest right now, cars from the same convoy would arrive out of order.
3. So the plaza uses a rule instead: take the number plate, add up the digits, and send the car to lane number remainder-of-that.
4. Every car from the same fleet, with related plates, ends up in the same lane, in order. Different fleets spread evenly across all eight.
5. Where this comparison breaks: a lane that closes forces a re-computation, and with a naive rule that reshuffles everybody, not just the affected cars. Real networks use hashing schemes designed to limit that damage.

26.11.3 PLAIN - a worked example

1. The fields used are called the **five-tuple**: source address, destination address, protocol number, source port and destination port.
2. For the reader's connection those are `192.168.0.14`, `20.207.73.82`, protocol 6, some ephemeral source port, and destination port 443.

3. Suppose a router has four equal paths. It hashes the five-tuple to a number and takes that number modulo 4.
4. A second connection from the same laptop to the same server uses a different source port, so a different hash, so possibly a different path.

Flow	Differs by	Same path?
Same TCP connection	nothing	Yes
New connection, same host	source port	Not necessarily
Ping probes	protocol is ICMP	Different hash space
Different destination	dest address	Not necessarily

5. Now look at the trace again. Hop 8 shows `be23.rwa02.bom01.ntwk.msn.net` and `po22.rwa04.bom01.ntwk.msn.net`. Two different Mumbai routers answered probes with the same TTL.
6. That means the probes took two different equal-cost paths through Mumbai. Both are correct answers.

26.11.4 PLAIN - what is really happening inside

1. There are two ways to spread traffic, and the difference matters enormously.
2. **Per-flow** hashing keeps a conversation on one link. Packets arrive in order. Each individual conversation is limited to the speed of one link.
3. **Per-packet** spreading sends each packet down whichever link is free. It uses capacity perfectly and reorders packets badly.
4. Reordering hurts TCP. Three out-of-order acknowledgements look exactly like a lost packet, so TCP slows down even though nothing was lost.
5. That is why per-flow is the default nearly everywhere, and per-packet is used only in controlled environments.
6. Bundling several physical links into one logical link, which is what `ae`, `be` and `po` refer to, uses the same hashing idea one layer lower down.

26.11.5 TECHNICAL - the engineer's version

1. ECMP is described in RFC 2991, Multipath Issues in Unicast and Multicast Next-Hop Selection, and RFC 2992, Analysis of an Equal-Cost Multi-Path Algorithm, both from November 2000.
2. RFC 2992 analyses hash-threshold assignment, chosen specifically because it disturbs fewer existing flows when the number of paths changes than a naive modulo does.
3. Typical hash inputs by layer:

Layer	Fields hashed
L2 bundle	Source and dest MAC
L3 only	Source and dest IP
L3 plus L4	The five-tuple
Tunnelled	Outer header, or entropy label

4. Tunnels break flow spreading. If thousands of flows share one outer header, the hash sees one flow. MPLS entropy labels and the UDP source port in VXLAN and in QUIC exist to restore variety.
5. Traceroute has a specific defence. The Paris traceroute technique keeps the flow identifier constant across probes so that all probes follow one path, which makes multipath fan-out disappear and reveals the true single path.
6. On macOS and Linux, `traceroute -T -p 443` fixes the destination port and protocol, which alone removes much of the scatter.

26.11.6 WORDS - remember these

1. ECMP — using several equally good paths at once — equal cost multipath forwarding across next hops with identical metrics.
2. Five-tuple — the five fields that identify a conversation — source IP, destination IP, protocol, source port and destination port.
3. Flow hashing — arithmetic that pins a conversation to one link — hashing selected header fields to choose among equal cost next hops.
4. Per-flow versus per-packet — keep a conversation together, or spread every packet — order-preserving versus capacity-maximizing load distribution.
5. Link aggregation — several cables acting as one — bundling physical links, called `ae`, `be` or `po` depending on the vendor.

26.12 Quality of service and the real world

26.12.1 PLAIN - in simple words

1. When packets arrive at a router faster than they can leave, they wait in a queue. That queue is called a **buffer**.
2. A small buffer smooths out short bursts. That is useful.
3. A large buffer holds packets for a long time before sending them. That is harmful, and it is why a fast connection can feel slow.
4. The packet is not lost. It is delayed, sometimes by a whole second, which for a video call or a game is worse than losing it.
5. This problem has a name: **bufferbloat**.
6. When a buffer finally fills completely, the simplest routers throw away whatever arrives next. That is called **tail drop**.
7. Better routers drop or mark a few packets early, on purpose, before the queue gets long, to tell senders to slow down sooner.
8. That deliberate early dropping is called **active queue management**.

26.12.2 PLAIN - a picture in your head

1. Think of a single checkout in a shop with a very long roped queue behind it.
2. Adding more rope does not make the checkout faster. It only lets more people stand in line, so everybody waits longer.
3. Somebody who wants one item now waits behind a trolley shopper's whole load.
4. The fix is not more rope. It is a shorter permitted queue, and a rule that turns people away early rather than letting the line grow without limit.
5. Where this comparison breaks: shoppers who are turned away go home. A sender whose packet is dropped simply slows down and tries again, which is exactly the message the drop was meant to send.

26.12.3 PLAIN - a worked example

1. Take a home connection with a 10 Mbit/s upload and a 1 megabyte buffer in the modem, which is not unusual.
2. One megabyte is 8,000,000 bits. At 10,000,000 bits per second, a full buffer takes 0.8 seconds to drain.
3. So while a large upload is running, every other packet from that house waits up to 800 milliseconds behind it.
4. A video call needs round trips under about 150 milliseconds to feel natural. The call breaks even though the link is not full.

Uplink	Buffer	Worst added delay
10 Mbit/s	1 MB	800 ms
10 Mbit/s	64 KB	51 ms
100 Mbit/s	1 MB	80 ms

5. Reducing the buffer from 1 MB to 64 KB cuts the worst case from 800 milliseconds to 51, on the same link, with no extra bandwidth.

26.12.4 PLAIN - what is really happening inside

1. Two different tools are often confused: shaping and policing.
2. **Shaping** delays traffic to make it fit a target rate. Packets queue and come out smoothly. Nothing is thrown away unless the queue overflows.
3. **Policing** simply discards anything above the target rate. There is no queue, so there is no added delay, but there is more loss.
4. Shaping is kinder to TCP. Policing is cheaper to implement and common on provider edges.
5. Marking is a third idea. The DSCP field in the IP header lets a sender label a packet as more or less urgent.
6. The honest version: your marking usually does not survive. Most networks rewrite or zero the DSCP field at their boundary, because they will not let an outsider decide their queueing.

26.12.5 TECHNICAL - the engineer's version

1. Jim Gettys named bufferbloat in 2010 and wrote it up as Bufferbloat: Dark Buffers in the Internet in ACM Queue in 2011.
2. Kathleen Nichols and Van Jacobson published CoDel, Controlling Queue Delay, in ACM Queue in 2012. It became RFC 8289, and the flow-queueing variant fq_codel became RFC 8290, both in January 2018.
3. CoDel does not target a queue length. It targets a queue delay, with a default target of 5 milliseconds measured over a 100 millisecond interval.
4. fq_codel adds per-flow queues with round-robin service, so a bulk transfer cannot delay a DNS lookup or a video call sharing the same link.
5. Earlier active queue management used Random Early Detection, from the 1993 paper by Sally Floyd and Van Jacobson, which drops with rising probability as the average queue grows. RED needs tuning; CoDel does not.
6. ECN, RFC 3168, lets a router set a bit instead of dropping. The receiver echoes it, the sender slows down, and nothing is lost.

DSCP name	Value	Typical use
CS0, default	0	Everything unmarked
AF41	34	Interactive video
EF	46	Voice
CS6	48	Network control

7. Tools: `tc qdisc` on Linux shows and sets the queue discipline, and `fq_codel` is the default on most current distributions. The `flent` test suite measures bufferbloat directly.
8. Whether providers should deprioritize particular traffic is a policy argument, not a technical one, and experts disagree. In India, the regulator TRAI barred discriminatory pricing of data services in February 2016 and net neutrality rules followed in 2018.

26.12.6 WORDS - remember these

1. Buffer — the waiting room for packets — memory holding packets queued for an outgoing interface.
2. Bufferbloat — a fast link that feels slow — excessive latency caused by oversized unmanaged buffers.
3. Tail drop — throw away whatever arrives when full — the default queue discipline, dropping arrivals once the buffer is full.
4. AQM — dropping early on purpose — active queue management, signalling congestion before the buffer fills.
5. fq_codel — many short queues, each kept short — the flow queue CoDel scheduler, RFC 8290, targeting 5 ms of queue delay.
6. Shaping — delay traffic to fit a rate — queueing excess traffic for later transmission.

26.13 Reading a packet capture

26.13.1 PLAIN - in simple words

1. Everything in this chapter can be watched directly. You do not have to guess.
2. A **packet capture** is a recording of the actual frames going in and out of a network card.
3. The command line tool is `tcpdump`. The graphical tool is Wireshark. They read the same file format.
4. There are two kinds of filter, and mixing them up wastes hours.
5. A **capture filter** decides what gets recorded. It runs inside the kernel and what it rejects is gone forever.
6. A **display filter** decides what you are shown from an existing recording. Nothing is lost, and you can change it as often as you like.

26.13.2 PLAIN - a picture in your head

1. A capture filter is the security guard at the door of a building, deciding who is allowed in at all.
2. A display filter is you, later, searching the visitor book for one name.
3. If the guard turned somebody away, no amount of searching the book will find them. The record simply does not exist.
4. Where this comparison breaks: the guard is fast and free, and letting everybody in has a real cost. On a busy link, capturing everything can fill a disk in minutes and drop packets while doing so.

26.13.3 PLAIN - a worked example

1. Here is what the reader's failure looks like on the wire.

```
$ sudo tcpdump -ni en0 host 20.207.73.82
09:14:02.118 IP 192.168.0.14.51422 > 20.207.73.82.443:
   Flags [S], seq 1829384756, win 65535, length 0
09:14:03.121 IP 192.168.0.14.51422 > 20.207.73.82.443:
   Flags [S], seq 1829384756, win 65535, length 0
09:14:05.128 IP 192.168.0.14.51422 > 20.207.73.82.443:
   Flags [S], seq 1829384756, win 65535, length 0
09:14:09.142 IP 192.168.0.14.51422 > 20.207.73.82.443:
   Flags [S], seq 1829384756, win 65535, length 0
```

2. Read the four things that matter.
3. Every line is outbound. Nothing at all comes back. Not a SYN-ACK, not a reset, not an ICMP message.
4. The sequence number is identical every time, so these are retransmissions of one SYN, not four separate attempts.
5. The gaps are 1, 2 and 4 seconds. That is TCP's exponential backoff, which is the operating system trying harder and harder before giving up.
6. The source port `51422` stays the same, confirming one connection attempt.
7. That capture, on its own, PROVES the request left the laptop and no answer returned. It does not prove where the packet died.

8. Compare with a refused connection, which replies at once with Flags [R], a reset, usually within milliseconds. The reader saw nothing of the kind.

26.13.4 PLAIN - what is really happening inside

1. Capturing works by putting the network card driver into a mode where it hands a copy of every frame to a special kernel facility.
2. That facility applies the capture filter, compiled into a tiny program, and copies only matching frames into a ring buffer for the tool to read.
3. Because the filter runs in the kernel, a narrow capture filter costs almost nothing even on a fast link.
4. Two cautions. On Wi-Fi you normally see only your own traffic unless the card is put into monitor mode. And on a switch you see only frames addressed to you, unless a port is configured to mirror others.
5. Encrypted payloads stay encrypted. You can see that a TLS handshake happened, its size and its timing, but not the HTTP request inside it.

26.13.5 TECHNICAL - the engineer's version

1. `tcpdump` was written in 1988 at Lawrence Berkeley Laboratory by Van Jacobson, Craig Leres and Steven McCanne. The filtering engine underneath it, the BSD Packet Filter, was described by McCanne and Jacobson in 1993.
2. Wireshark began as Ethereal, released by Gerald Combs in 1998, and was renamed Wireshark in 2006 after a trademark problem.
3. Capture filters use BPF syntax. Display filters use Wireshark's own syntax. They are not interchangeable, and this trips up nearly everyone once.

```
sudo tcpdump -ni en0 -s0 -w gh.pcap \
  'host 20.207.73.82 and tcp port 443'
tcpdump -r gh.pcap -vvn
```

4. Useful display filters in Wireshark for this fault: `ip.addr == 20.207.73.82`, `tcp.flags.syn == 1 && tcp.flags.ack == 0`, `tcp.analysis.retransmission`, and `icmp`.
5. `-s0` captures whole frames rather than truncating them. `-w` writes a pcap file. Analysis should always be done on a file, never on live scrolling text.
6. A checklist for diagnosing silence, in order:
    1. Is there an outbound SYN at all, and to the right address?
    2. Is anything at all coming back, including ICMP?
    3. Are the retransmissions identical, or is the stack trying new ports?
    4. Does a capture on another network, such as mobile data, differ?
7. Tools worth having beside tcpdump: `mtr` for continuous per-hop loss, `dig` for name resolution, `curl -v --trace-time` for per-stage timing, and `tshark` for scripted analysis of a saved capture.

26.13.6 WORDS - remember these

1. Packet capture — a recording of real traffic — a stored copy of frames read from an interface, normally in pcap or pcapng format.
2. Capture filter — decides what is recorded — a BPF expression evaluated in the kernel before frames are copied to userspace.
3. Display filter — decides what is shown — a Wireshark expression applied to an already captured set of packets.
4. Promiscuous mode — the card keeps frames not addressed to it — an interface setting required to see other stations' traffic.
5. Retransmission — the same data sent again — a repeat of an unacknowledged segment, identifiable by an identical sequence number.

26.98 Common wrong ideas

1. Wrong: packets travel down a fixed path that is set up when the connection starts. Right: nothing is set up. Each router decides independently, per packet, and the path can change mid-conversation without anyone being told.
2. Wrong: the internet routes around damage automatically and instantly. Right: it routes around damage eventually, if an alternative exists and if policy allows it. Convergence takes seconds inside one network and can take minutes across BGP, during which traffic is lost.
3. Wrong: a traceroute shows the return path. Right: it shows one address per hop on the way out, learned from replies that came back by an unknown route. The return path is invisible from one end.
4. Wrong: more hops means slower. Right: hop count and latency are only loosely related. Twelve hops inside one well-run backbone can beat four hops over a congested link or a satellite.
5. Wrong: stars at the end of a traceroute prove a fault. Right: many routers rate limit or block ICMP replies. The reader's hops 13 to 20 showing `* * *` is consistent with a perfectly healthy path.
6. Wrong: seeing two or three addresses for one hop means something is broken. Right: it means parallel links are in use, which is normal and desirable.
7. Wrong: BGP picks the shortest path. Right: BGP picks the path its operator's policy prefers. Local preference, which is pure business policy, is compared before AS path length, and physical distance is never compared at all.
8. Wrong: my ISP can see which pages I loaded because it carries my packets. Right: it sees addresses, sizes and timing. With TLS it does not see the content, though it can often infer a great deal from the pattern.

26.99 Chapter summary in 20 lines

1. Data is split into packets for fairness, cheap error recovery, small buffers and the ability to interleave many conversations on one wire.
2. Circuit switching reserves a path and wastes capacity. Packet switching reserves nothing and accepts variable delay. The internet chose the second.
3. A packet is a header plus a payload, and the payload is normally another whole packet with its own header.
4. A 512 byte HTTPS request from the reader's laptop becomes a 604 byte Ethernet frame, which is 15.2 percent overhead.
5. The IPv4 header is 20 bytes and 13 fields, unchanged since RFC 791 in 1981.
6. TTL stops packets circling forever. Decrementing it forces the header checksum to be recomputed, which routers do incrementally.
7. Every link has an MTU. Oversized packets are fragmented, rejected with an ICMP message, or silently dropped, and the third case is the painful one.
8. IPv6 never fragments in the network. Modern IPv4 senders set the Do Not Fragment bit and discover the path MTU for themselves.
9. A router receives a frame, strips it, looks up the destination, decrements TTL, fixes the checksum, resolves the next hop and builds a brand new frame.
10. MAC addresses change at every hop. IP addresses do not, unless a device is performing NAT.
11. A routing table is matched by longest prefix. `20.207.73.0/24` beats `20.192.0.0/10` beats `0.0.0.0/0` for the address `20.207.73.82`.
12. The reader's default route, `0.0.0.0/0` through `192.168.0.1`, carried every one of these packets out of the flat.
13. Inside a network, routes are learned by RIP, OSPF, IS-IS or EIGRP. Link state protocols flood a map and run Dijkstra's algorithm on it.
14. Between networks, routes are learned by BGP, which carries a full AS path and chooses by policy, with local preference outranking path length.

15. `20.207.73.82` belongs to AS8075, Microsoft, inside the route `20.192.0.0/10`, which is why hop 7 onwards reads `ntwk.msn.net`.
16. BGP has no built-in way to verify an announcement. RPKI adds signed origin authorizations, and about 69 percent of IPv4 prefix-origin pairs were valid in August 2026, but only about a quarter of networks enforce checking.
17. Nobody knows the whole path in advance. Each router knows only the next hop, and the return path is chosen by different networks entirely.
18. Multiple addresses at hops 5, 6, 8, 9 and 11 are equal cost multipath at work, selected by hashing the five-tuple, and are healthy rather than faulty.
19. Oversized buffers turn loss into delay and make fast links feel slow. Short managed queues, such as fq_codel, fix it without more bandwidth.
20. A capture showing four identical SYNs at 1, 2 and 4 second gaps with no reply of any kind PROVES the request left and nothing came back, and SUGGESTS a silent drop somewhere on that path.

27

CHAPTER

KB

Traceroute Decoded — Reading Your Own Path

27.0 What this chapter gives you

1. You will be able to explain why there is no "show me the route" command in the internet protocol, and why traceroute is a trick rather than a feature.
2. You will be able to describe the time-to-live field exactly: how many bits it has, what it counts, and what happens at each router.
3. You will be able to guess which operating system sent you a packet by looking at one number in it, and to say why that guess can be wrong.
4. You will be able to draw the full sequence of probes and replies that produces a traceroute, and say what each printed line really is.
5. You will be able to choose between the three main traceroute methods and say which one gets furthest through modern filtering, and why.
6. You will be able to explain why one hop shows several addresses, and point at the exact hops in the reader's own trace where this happened.
7. You will be able to read a row of stars honestly, and to say out loud that stars at the end of a trace prove nothing on their own.
8. You will be able to read every hop of the reader's real trace, including every router name, city code and interface name.
9. You will be able to say precisely what a traceroute cannot tell you, which is a longer and more important list than what it can.
10. You will be able to run a much better traceroute next time, with the exact flags, and to get a second opinion from the far end of the path.

27.1 Traceroute is a clever trick, not a feature

27.1.1 PLAIN – in simple words

1. When you send data across the internet, it passes through many machines before it arrives.
2. Each of those machines is a **router**: a box whose only job is to take a packet in one door and push it out the right other door.
3. The internet protocol has no field that records the route. Packets do not carry a list of where they have been.
4. Routers do not report to you. A router does not know or care who you are. It looks at the destination, picks a door, and forgets you instantly.
5. Traceroute gets the answer anyway, by cheating.
6. It sends packets that are designed to fail. It sends them so they die at the first router, then the second, then the third.

7. Every time a packet dies, the router that killed it sends back a short complaint. That complaint has the router's own address on it.
8. Nobody meant for this to be possible. It is a side effect of two safety rules that were written for completely different reasons.

27.1.2 PLAIN - a picture in your head

1. Imagine you want to know the route your letters take to a friend in another city, and the postal service will not tell you.
2. But there is a rule: if a letter has been handled too many times without being delivered, an office must destroy it and post you a note saying so.
3. The note is stamped with that office's own name and town, because the rule also says the office must identify itself.
4. So you write a letter and mark it "destroy after one handling". Back comes a note from your local sorting office. You now know step one.
5. Twenty letters later you have the whole route, and you never once asked anyone a question they were allowed to refuse.

Where this comparison breaks: a real postal route is stable, so twenty letters would follow the same path. On a network each of your twenty probes can legitimately take a different road. A traceroute is a sketch assembled from many separate journeys, not a photograph of one. That single fact explains most of the confusing things in this chapter.

27.1.3 PLAIN - a worked example

1. Here is the reader's own trace, exactly as their machine printed it. We are going to spend most of this chapter on these twenty lines.

```
 1  192.168.0.1
 2  172.31.0.17
 3  137.97.29.249
 4  172.26.22.235
 5  172.16.18.33 / 172.26.14.75 / 172.26.22.234
 6  172.26.14.75 / 172.16.18.33
 7  ae66-0.del01-96cbe-1b.ntwk.msn.net (104.44.196.187)
 8  be23.rwa02.bom01.ntwk.msn.net (104.44.55.163)
    po22.rwa04.bom01.ntwk.msn.net (104.44.51.55)
 9  be1010.owr02.bom01.ntwk.msn.net (104.44.53.119)
    be1013.owr01.bom01.ntwk.msn.net (104.44.53.47)
10  be5.ibr02.pnq21.ntwk.msn.net (104.44.31.62)
11  ae104-0.rwa03.pnq21.ntwk.msn.net (104.44.20.52)
    be1.ibr01.pnq20.ntwk.msn.net (104.44.16.51)
12  ae106-0.rwa04.pnq20.ntwk.msn.net (104.44.20.42)
13-20  * * *   (no reply)
```

2. Twenty lines. Twelve of them answered. Eight of them are silence.
3. Every single line is a death notice. Something threw the reader's packet away and sent back a note about it.
4. The router names on hops 7 to 12 were not sent by the routers either. Those came from a separate name lookup done afterwards by the reader's own machine, on the addresses in brackets.
5. So even the names are a second-hand fact. We will come back to that, because it matters when a name is wrong or missing.

27.1.4 PLAIN - what is really happening inside

1. There were two serious attempts to build a real "record the route" feature into the internet, and both failed for good reasons.

2. The first is called the **record route option**. It is part of the original internet protocol from 1981. A packet can carry a small list, and each router writes its address into the list.
3. It does not work in practice. The space for options in the packet header is only 40 bytes, which holds at most nine addresses. Real paths are longer.
4. The second attempt was a proper traceroute protocol, published in 1993. It would have used one packet instead of dozens. Almost nobody implemented it.
5. So the working method is the trick. It relies on two rules that every router already obeys for its own reasons.
6. Rule one: every packet carries a counter, and every router must reduce that counter by one. When it hits zero the packet is destroyed. This exists to stop packets circling forever when routing goes wrong.
7. Rule two: when a router destroys a packet for that reason, it must send a short error message back to whoever sent it. This exists so the sender can find out its packets are dying.
8. Traceroute exploits both. It sets the counter deliberately low, so the packet dies on purpose at a chosen distance, and the error message that comes back carries the address of the killer.

27.1.5 TECHNICAL - the engineer's version

1. The IPv4 header defined in RFC 791, September 1981, is 20 bytes without options. There is no field in it that accumulates a path.
2. The Record Route option is IP option number 7, defined in RFC 791. Options live in the space between byte 20 and byte 60 of the header, so the whole option area is at most 40 bytes.
3. Packets with IP options are process-switched or punted to the control plane on most modern hardware routers, rather than handled by the forwarding silicon. Many operators drop them outright as a denial-of-service defence.
4. RFC 1393, "Traceroute Using an IP Option", by G. Malkin of Xylogics, January 1993, is an Experimental specification. It proposed a new IP option and a new ICMP message so that one packet could map a whole path.
5. The mechanism that actually works is emergent behaviour from two normative requirements. RFC 1812, "Requirements for IP Version 4 Routers", edited by Fred Baker, June 1995, section 5.3.1, requires that a router "MUST decrement the TTL field of every datagram by exactly one", and that when the result is zero the router MUST discard the datagram and MUST send an ICMP Time Exceeded message.
6. The complementary requirement is in RFC 792, "Internet Control Message Protocol", by Jon Postel, September 1981, which defines that Time Exceeded message as type 11, code 0.
7. Traceroute itself was written by Van Jacobson in 1987, from a suggestion by Steve Deering. The manual page has said for decades: "Implemented by Van Jacobson from a suggestion by Steve Deering." Guy Almes and Matt Mathis are credited with having had the same idea independently.
8. Van Jacobson worked in the Network Research Group at Lawrence Berkeley National Laboratory in California, which released traceroute along with tcpdump and other network tools.

Attempt	Year	Why it is not used
Record Route option	1981	Only 9 addresses fit
RFC 1393 traceroute	1993	Never deployed
TTL trick	1987	This is the one in use

9. Observe it with `traceroute` on macOS, Linux and BSD, `tracert` on Windows, `tracepath` on Linux, and `mtr` everywhere.

27.1.6 WORDS - remember these

1. Router — a box that passes packets along — a layer 3 forwarding device that makes a next-hop decision per packet from a routing table.

2. Traceroute — a tool that lists the machines on the way out — a utility that elicits ICMP Time Exceeded responses by manipulating the IP TTL field.
3. Probe — one test packet — a single datagram sent with a specific TTL and a specific identifier so its response can be matched to it.
4. Record Route option — an old idea for packets to log their own path — IP option 7 from RFC 791, limited to nine addresses by the 40-byte option area.
5. Emergent behaviour — a use nobody designed for — functionality arising from the interaction of unrelated normative requirements.

27.2 The time to live field, properly

27.2.1 PLAIN - in simple words

1. Every packet on the internet carries a small number called the **time to live**, almost always shortened to TTL.
2. It is one byte. That means it can hold any whole number from 0 to 255.
3. The name is misleading. It sounds like a clock. It is not a clock.
4. It counts hops, not seconds. A hop is one router handling the packet.
5. Every router that forwards the packet subtracts one from it.
6. When the number reaches zero, the packet is thrown away. It goes no further, no matter how important it was.
7. This is a safety belt. Without it, a mistake in the routing tables could make a packet loop between two routers forever, and the network would slowly fill up with immortal garbage.
8. Different computers start the number at different values. That turns out to be a small accidental fingerprint.

27.2.2 PLAIN - a picture in your head

1. Think of a bus ticket with a row of boxes on it, and a number printed on the front saying how many boxes are left.
2. Every time you board a bus, the driver punches a box and reduces the number by one.
3. When the number reaches zero the driver refuses you and puts you off the bus, wherever you happen to be.
4. The ticket does not expire at a time of day. It expires after a count of journeys. A slow bus and a fast bus cost exactly the same: one punch.
5. Now imagine the rule says the driver who puts you off must post a postcard to whoever bought your ticket, saying which stop it happened at.

Where this comparison breaks: a router that holds a packet for a whole second still only subtracts one. The 1981 specification really did intend a clock, in seconds, and asked routers to subtract at least one per second of holding time. No router implemented the clock part, and in 1995 the rule was rewritten to say plainly: subtract exactly one, every time. The name is a fossil.

27.2.3 PLAIN - a worked example

1. The reader's machine is a Mac. macOS starts outgoing packets with TTL 64.
2. Follow one packet from the reader's flat toward hop 12 of their own trace.

```
Packet leaves the Mac with TTL = 64

  at 192.168.0.1      64 -> 63   forward
  at 172.31.0.17      63 -> 62   forward
  at 137.97.29.249    62 -> 61   forward
  at 172.26.22.235    61 -> 60   forward
  at hop 5 router     60 -> 59   forward
```

```
at hop 6 router     59 -> 58   forward
at 104.44.196.187   58 -> 57   forward   (Delhi)
at 104.44.55.163    57 -> 56   forward   (Mumbai)
at 104.44.53.119    56 -> 55   forward   (Mumbai)
at 104.44.31.62     55 -> 54   forward   (Pune)
at 104.44.20.52     54 -> 53   forward   (Pune)
at 104.44.20.42     53 -> 52   forward   (Pune)
```

3. Now run the trick backwards. Suppose a reply comes back to you with a TTL of 52 in it. You guess it started at 64 and crossed 12 routers.

Received TTL	Likely start	Hops crossed
52	64	12
118	128	10
245	255	10
63	64	1

4. From that you can often guess what kind of machine sent it. A reply arriving with 118 was probably sent by Windows. One arriving with 52 was probably sent by something built on Unix, such as Linux or macOS.
5. This is a guess, not a measurement. Say so out loud when you use it.

27.2.4 PLAIN - what is really happening inside

1. The TTL sits at a fixed place in the packet header, so a router can find it without understanding anything else about the packet.
2. On arrival the router reads it. If it is already 0 or 1, the packet cannot be forwarded, because forwarding would require subtracting one and reaching zero or below.
3. In that case the router drops the packet and builds an error message to send back to the source address printed on the dead packet. Otherwise it subtracts one and forwards.
4. There is a hidden cost here. The header carries a checksum, a small number that proves the header was not corrupted in transit.
5. Change the TTL and the checksum is now wrong, so the router must fix the checksum too, on every packet, at full line rate.
6. Note carefully who does the subtracting. Only devices that **forward** the packet. A plain network switch does not touch the TTL, because it works at a lower level and is not a router. This is why a large office network with many switches still shows as one hop.

27.2.5 TECHNICAL - the engineer's version

1. In IPv4 the TTL is an 8-bit unsigned field at byte offset 8 of the header, the ninth byte, immediately before the protocol field. Range 0 to 255.
2. RFC 791, September 1981, described it in time units and required a decrement of at least one per hop and per second of queuing.
3. RFC 1812, June 1995, section 5.3.1 replaced that with the flat rule: a router MUST decrement by exactly one. The time semantics are dead.
4. In IPv6 the field was renamed to **Hop Limit**, still 8 bits, at byte offset 7 of the 40-byte fixed header. The rename admits what the field always was.
5. The IANA IP Parameters registry states: "The current recommended default time to live (TTL) for the Internet Protocol (IP) is 64", citing RFC 791 and RFC 1122.
6. Linux exposes the default as the sysctl `net.ipv4.ip_default_ttl`, documented as "Default value of TTL field (Time To Live) for outgoing (but not forwarded) IP packets", with a documented default of 64.

7. Common starting values, all of them product defaults rather than standards, and all of them changeable by an administrator:

System	Start TTL	Kind of fact
Linux	64	kernel default
macOS and FreeBSD	64	kernel default
Windows, modern	128	product default
Cisco IOS and IOS-XR	255	product default
IANA recommendation	64	RFC 791, RFC 1122

8. Guessing the sender from a received TTL is called passive OS fingerprinting and is implemented in tools such as `p0f` and in the `ttl` matcher of several firewalls. It is weak evidence, for four reasons.
9. First, a tunnel changes the count. The reader's machine showed several `utun` interfaces, which are point-to-point tunnel devices used by VPN and similar software. Traffic through a tunnel gets a different hop count.
10. On macOS you can read the local default with `sysctl net.inet.ip.ttl`, and macOS `traceroute` uses that same value as the default maximum hop count, documented as "net.inet.ip.ttl hops (the same default used for TCP connections)".

27.2.6 WORDS - remember these

1. TTL — a hop counter carried in every packet — an 8-bit IPv4 header field decremented by exactly one at each forwarding router, per RFC 1812 5.3.1.
2. Hop — one router handling the packet — one layer 3 forwarding decision.
3. Hop Limit — the IPv6 name for TTL — the 8-bit field at offset 7 of the IPv6 fixed header, semantically identical.
4. Header checksum — a small proof the header is intact — the 16-bit ones complement checksum over the IPv4 header only, updated incrementally.
5. Passive OS fingerprinting — guessing the sender's system from clues — the inference of the originating stack from TTL, window size and option order.

27.3 The mechanism, step by step

27.3.1 PLAIN - in simple words

1. Traceroute does the same small thing over and over, with one number changed each time.
2. Step one. Send a packet toward the final destination, but set its hop counter to 1.
3. The first router receives it, subtracts one, gets zero, and must throw it away. It sends back an error message that says "your packet ran out of hops at my place".
4. That error message has the router's own address as its sender. That is how the reader learned that hop 1 is `192.168.0.1`.
5. Step two. Send another packet, exactly the same, but with the hop counter set to 2. It survives the first router and dies at the second.
6. The second router complains, and now the reader knows hop 2 is `172.31.0.17`. Repeat with 3, then 4, then 5, until you reach the destination or give up.
7. Three packets are sent for each hop by default, not one, so a slow or lost probe does not hide a router.
8. Every line you see on screen is therefore a different packet. The trace is not one journey observed. It is dozens of separate short journeys.

27.3.2 PLAIN - a picture in your head

1. Picture a long corridor of doors, one behind the other, and you standing at the near end. You cannot see past the first door.
2. You have a supply of paper darts. Each dart has a number written on it.
3. The rule of the corridor is that each person who catches a dart writes one less on it and throws it to the next person, unless the number would go to zero.
4. If the number would go to zero, that person keeps the dart, writes their own name on a card, and walks the card back to you.
5. So you throw a dart marked 1 and the nearest person hands you a card. Throw one marked 2 and the second person does. Twenty darts give you the whole list in order.
6. You also time each card. You start a stopwatch when the dart leaves your hand and stop it when the card arrives.

Where this comparison breaks: the person walking the card back may take a completely different corridor, which you never see, so the stopwatch measures a loop and not a distance. Also, some people are under orders never to walk cards back, and some are so busy they return only one card a second. Both turn into stars on your screen.

27.3.3 PLAIN - a worked example

1. Here is the full sequence for the first three hops of the reader's trace. Nine packets out, nine replies in.

Probe	TTL sent	Who replied
1, 2, 3	1	192.168.0.1
4, 5, 6	2	172.31.0.17
7, 8, 9	3	137.97.29.249

2. Notice that the destination address on all nine packets was the same: the GitHub address `20.207.73.82`. Only the TTL changed.
3. Now the whole process as a diagram. `X` marks where a probe dies.

```
YOU              R1        R2        R3          TARGET
 |               |         |         |            |
 |--TTL=1------->X         |         |            |
 |<--Time Exceeded         |         |            |
 |   src = R1    |         |         |            |
 |               |         |         |            |
 |--TTL=2------->|-------->X         |            |
 |<--Time Exceeded---------|         |            |
 |   src = R2    |         |         |            |
 |               |         |         |            |
 |--TTL=3------->|-------->|-------->X            |
 |<--Time Exceeded-------------------|            |
 |   src = R3    |         |         |            |
 |               |         |         |            |
 |--TTL=4------->|-------->|-------->|----------->|
 |<--reply from the target itself, different type-|
 |               |         |         |            |
```

4. The last line is different on purpose. When a probe finally survives to the destination, the destination does not send a Time Exceeded message, because the packet did not run out of hops. It sends something else, and what it sends depends on which traceroute method you used.
5. Timing. Each of the three probes at each hop is timed on its own, which is why a normal traceroute prints three separate millisecond figures per line.

6. The reader's captured trace does not show those figures. It records only the addresses. We will come back to that gap in section 27.7, because it is a real limitation of the evidence they collected.

27.3.4 PLAIN - what is really happening inside

1. The error message that comes back is not empty. It carries a copy of the dead packet inside it.
2. That copy is the key to the whole system. Without it, traceroute could not tell which of its many probes a given reply belongs to.
3. Replies arrive from different routers, in any order, some late, some never. Each reply must be matched to exactly one probe.
4. The trick is that the router quotes the dead packet back at you. Inside the error message you find the original header, including whatever identifying number traceroute put in it.
5. Classic traceroute puts that identifier in the destination port number. It uses a different port for every single probe.
6. So probe number 7 goes to port 33440, and when a reply comes back quoting port 33440, traceroute knows this is the answer to probe 7, sent at a known moment, with a known TTL.
7. This clever identifier scheme has one bad side effect, and it causes most of the confusion in the reader's trace. Changing the port for every probe changes what load-balancing routers see, so different probes get sent down different links. Section 27.5 is entirely about that.

27.3.5 TECHNICAL - the engineer's version

1. The response of interest is ICMP type 11, code 0. RFC 792 gives the code meanings verbatim as "time to live exceeded in transit" for code 0 and "fragment reassembly time exceeded" for code 1.
2. In IPv6 the same event is ICMPv6 type 3, code 0, "Hop limit exceeded in transit", defined in RFC 4443, March 2006, by Conta, Deering and Gupta.
3. The ICMP error payload contains the IP header of the offending datagram plus at least the first 8 bytes of its payload, which is enough to capture the UDP or TCP source and destination ports, or the ICMP identifier and sequence number.
4. RFC 1812 section 4.3.2.3 upgraded that to say the error should contain as much of the original datagram as possible without exceeding a total ICMP message size of 576 bytes.
5. Source address selection matters for reading a trace. RFC 1812 section 4.3.2.4 requires the source address of a router-originated ICMP message to be an address of the interface over which the ICMP message is transmitted.
6. On a point-to-point link, that interface is the same one the probe arrived on. This is why traceroute conventionally shows the **ingress** interface address of each router, not a loopback address or a router identifier.
7. That single fact explains the interface names in the reader's hops 7 to 12. `ae66-0` and `be23` are names of the specific interfaces the probes came in on, not names of the routers.
8. The macOS manual page states the defaults as "-q nqueries: Set the number of probes per ttl to nqueries (default is three probes)", "-p port: default is 33434", and "-w: default 5 sec.".

Element	Classic default	Where set
Probes per hop	3	`-q`
Base UDP port	33434	`-p`
Wait per probe	5 s	`-w`
Max hops	64 on macOS	`-m`

27.3.6 WORDS - remember these

1. ICMP — the internet's error and status messaging — Internet Control Message Protocol, RFC 792, IP protocol number 1.

2. Time Exceeded — the "your packet ran out of hops" message — ICMP type 11 code 0 in IPv4, ICMPv6 type 3 code 0 in IPv6.
3. Quoted packet — the copy of the dead packet inside the error — the original IP header plus at least 8 payload bytes, used for demultiplexing.
4. Ingress interface — the door the packet came in through — the interface whose address a router normally uses as the ICMP error source.
5. Round-trip time — how long the there-and-back took — the measured interval between probe transmission and matching ICMP response reception, per probe.

27.4 The variants, and which one to use

27.4.1 PLAIN - in simple words

1. The trick with the hop counter works no matter what kind of packet you send. The counter is in the outer wrapper, which every packet has.
2. So there are several traceroutes, differing only in what they put inside the wrapper.
3. The original sends a small packet to a strange, unused port number on the destination. This is the classic method and it is still the default on Mac and Linux.
4. The Windows version sends a ping instead. A ping is the simplest possible "are you there" message.
5. The third version sends the first packet of a real connection attempt, aimed at a real service port, such as the one used by secure websites.
6. That second difference is the whole point. Networks are much more willing to pass traffic that looks like normal web traffic than traffic that looks like a diagnostic tool.
7. For the reader's problem, which was a web connection failing, the third version is clearly the right one, and it is not the default.

27.4.2 PLAIN - a picture in your head

1. Imagine three people trying to reach the same office in a guarded building.
2. The first wears a plain uniform nobody recognizes, and asks for room 33434, which does not exist. Guards find this suspicious and often turn them away.
3. The second shouts "hello, is anyone there?" from the doorway. Guards at many buildings have standing orders to ignore that shout entirely.
4. The third walks in wearing normal clothes and asks for the reception desk, room 443, which is the room everyone visits all day long.
5. The third person gets much further, because they look exactly like the thousands of legitimate visitors the guards are there to let in.

Where this comparison breaks: the guards are not usually deciding that they dislike diagnostics. Very often nobody blocked anything, and the difference is simply that the third kind of traffic is what the path is built and tuned to carry. Also, asking for reception has a real cost: you make the far end start work on a connection it will never finish.

27.4.3 PLAIN - a worked example

1. The three commands, for the reader's exact target, on macOS.

```
# 1. classic UDP, the default. No special permission needed.
traceroute 20.207.73.82

# 2. ICMP echo, the same method Windows tracert uses.
sudo traceroute -I 20.207.73.82

# 3. TCP to port 443, the one to use for a web problem.
sudo traceroute -P TCP -p 443 20.207.73.82
```

2. What each looks like at the moment it reaches the destination.

Method	Probe sent	Success reply
UDP classic	UDP to port 33434+	Port Unreachable
ICMP echo	Echo Request	Echo Reply
TCP to 443	TCP SYN to 443	SYN-ACK or RST

3. Notice that the classic method succeeds by provoking an error. It picks a port nothing is listening on precisely so the destination answers "nothing here", which proves you reached it.
4. The TCP method succeeds by getting an actual answer from an actual service. If the destination replies at all, you have proven far more than a map.

27.4.4 PLAIN - what is really happening inside

1. The important idea is that the intermediate hops behave identically in all three cases. Time Exceeded is generated for any packet whose counter runs out, regardless of what is inside.
2. But whether a probe reaches a middle hop at all depends on every device before it.
3. A firewall five hops along that drops odd UDP traffic will make everything past it look dead, even though the routers there are perfectly healthy.
4. Swap to TCP port 443 and the same probes sail through, and the hops appear.
5. There is one more family worth knowing: tools that repeat the whole trace continuously and build statistics, rather than running once and stopping.
6. Running once gives you a snapshot. Running for two minutes tells you whether a hop loses one packet in a hundred, which a snapshot can never show.

27.4.5 TECHNICAL - the engineer's version

1. UDP traceroute, the 1987 Van Jacobson design, sends UDP datagrams to destination ports starting at 33434 and increasing by one per probe, giving the familiar 33434 to 33534 window.
2. Termination is by ICMP type 3 code 3, Destination Unreachable, Port Unreachable, from the target's own stack.
3. ICMP traceroute sends ICMP Echo Request, type 8 code 0, and terminates on Echo Reply, type 0 code 0. This is the default of Windows `tracert`, and is available on macOS as `-I`, documented as "Use ICMP ECHO instead of UDP datagrams. (A synonym for -P icmp)".
4. TCP traceroute sends a TCP SYN to a chosen port, typically 80 or 443, and terminates on SYN-ACK from an open port or RST from a closed one.
5. The macOS manual page documents `-P proto` as "Send packets of specified IP protocol. The currently supported protocols are: UDP, TCP, GRE and ICMP". Raw socket creation means `-I` and `-P TCP` need root.
6. Why TCP to 443 gets further, in order of practical importance:

Reason	Effect on UDP or ICMP
Stateful firewall policy	High UDP ports dropped
ICMP policy on edges	Echo dropped or limited
Anycast and load balancers	Only serve real ports
QoS and CoPP classing	Diagnostics deprioritized

7. `mtr`, originally written by Matt Kimball around 1996 and maintained by Roger Wolff from October 1998, combines traceroute and ping. It re-probes continuously and reports per-hop loss percentage, best, worst, average and standard deviation. Modern builds support ICMP, UDP and TCP probes.

8. Paris traceroute was introduced in "Avoiding traceroute anomalies with Paris traceroute" by Brice Augustin, Xavier Cuvellier, Benjamin Orgogozo, Fabien Viger, Timur Friedman, Matthieu Latapy, Clemence Magnien and Renata Teixeira, at the Internet Measurement Conference in October 2006.
9. What it fixes: classic traceroute varies the destination port per probe, which changes the flow identifier that load balancers hash on, so consecutive probes take different paths and the printed path is a blend of several real paths that may never have existed as one route.
10. Paris traceroute holds the flow identifier constant across probes by varying a field the load balancer does not hash, classically the UDP checksum field, compensated so the checksum still validates.
11. Recommendation for the reader's exact situation, a failing HTTPS connection to `20.207.73.82`: use TCP to port 443. It is the only method whose probes are treated by every device on the path the same way the real failing traffic is treated. Anything else measures a different journey.

27.4.6 WORDS - remember these

1. Probe type — what kind of packet the tool sends — the transport and port combination that determines both filtering treatment and ECMP hashing.
2. Port Unreachable — "nothing is listening there" — ICMP type 3 code 3, the normal termination signal of UDP traceroute.
3. SYN — the first packet of a connection request — a TCP segment with the SYN flag set, opening the three-way handshake.
4. Paris traceroute — a traceroute that keeps probes looking identical — a variant holding the ECMP flow identifier constant across probes, from the 2006 Internet Measurement Conference paper.
5. MTR — a traceroute that never stops — a continuous hybrid of traceroute and ping reporting per-hop loss and latency statistics.

27.5 Why one hop shows several addresses

27.5.1 PLAIN - in simple words

1. Look at hop 5 of the reader's trace. It has three addresses on one line.
2. Neither is true. It is normal, healthy, and a sign of a well-built network.
3. Remember that three separate packets were sent for that hop. Three packets, three journeys, three answers.
4. Big networks do not join two places with one cable. They join them with several cables side by side, for capacity and for safety.
5. When several equally good roads exist, a router shares traffic across them.
6. So probe one went left, probe two went straight, probe three went right. Each landed on a different router, and each of those routers answered.
7. It means the network has spare capacity in that place. If one of those cables was cut, traffic would keep flowing on the others.
8. Five of the reader's hops show this. Hops 5, 6, 8, 9 and 11. That is not a fault, and it is not a sign of anything wrong.

27.5.2 PLAIN - a picture in your head

1. Imagine a wide river with three parallel bridges between the same two towns, and a traffic officer at the near end sending cars over all three.
2. You want to map the route. You send three cars, each told to stop at the far end of the first bridge and report which bridge they were on.
3. Now suppose the middle bridge is a little longer, with an extra toll booth halfway. Cars told to stop after the same number of booths stop in different places on different bridges.
4. That is why the reader's hop 11 shows two routers in two different cities. The parallel roads are not exactly the same length.

Where this comparison breaks: the officer chooses at random and a real router almost never does. It computes a number from the addresses and ports on each packet and uses that to pick a bridge, so two packets of one conversation always get the same bridge and cannot arrive out of order. Traceroute defeats that on purpose by changing the port on every probe.

27.5.3 PLAIN - a worked example

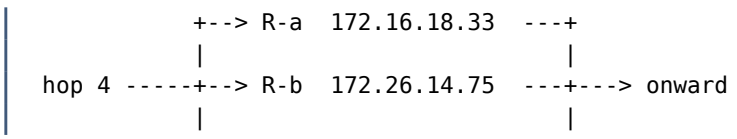
1. Here are the reader's five multi-address hops, exactly as recorded.

Hop	Addresses seen	What differs
5	three private	three separate routers
6	two private	same two as hop 5
8	rwa02, rwa04	two Mumbai routers
9	owr01, owr02	two Mumbai routers
11	rwa03, ibr01	two cities, Pune

2. **Hop 5.** `172.16.18.33`, `172.26.14.75` and `172.26.22.234`. All three probes took different links. Three answers, three addresses.
3. **Hop 6.** `172.26.14.75` and `172.16.18.33`. Both of these already appeared at hop 5.
4. That looks impossible until you remember the bridges. On one parallel road that router is five hops away; on another it is six.
5. **Hop 8.** Two Mumbai routers named `rwa02` and `rwa04`. Different machines in the same city doing the same job.
6. **Hop 9.** Two Mumbai routers named `owr01` and `owr02`. Same pattern, one step further in.
7. **Hop 11.** The most interesting one. One probe was answered by a router in Pune site 21, another by a router in Pune site 20.
8. So at the same hop count, two probes were in two different buildings. The two parallel paths through Pune are not the same length.

27.5.4 PLAIN - what is really happening inside

1. The technique is called **equal-cost multi-path**, usually shortened to ECMP. It means: when the routing table offers two or more equally good next steps, use all of them.
2. The router must decide, per packet, which of them to use, and there are two ways to decide.
3. The first way is **per-packet**: alternate strictly, one packet each. This uses the links perfectly evenly and it is almost never used, because packets of one conversation arrive out of order and confuse the receiver.
4. The second way is **per-flow**: work out which conversation the packet belongs to, and always send that conversation the same way. This is what virtually every network does.
5. To identify the conversation, the router takes a handful of fields from the packet and mixes them into a single number. Then it takes the remainder of that number divided by the number of links.
6. The fields are usually the source address, the destination address, the protocol, the source port and the destination port. Five things, so this is called the five-tuple.
7. Now the punchline. Classic traceroute changes the destination port on every single probe, because it uses the port as a serial number. Change the port, change the mixed number, change the chosen link.
8. Your web browser does the opposite. One TCP connection keeps the same five fields for its whole life, so it stays on one link the entire time.
9. This is the single most important thing to understand about reading a traceroute: the tool is not following your traffic. It is sampling all the roads your traffic could have taken.



```
            +--> R-c  172.26.22.234 ---+

probe 1 hashes to R-a, probe 2 to R-b, probe 3 to R-c
all three answer, so hop 5 prints three addresses
```

27.5.5 TECHNICAL - the engineer's version

1. ECMP is standard behaviour in every modern routing platform. Inside a single operator, it is usually driven by an interior gateway protocol such as OSPF or IS-IS installing multiple equal-metric next hops.
2. The hash input is configurable. Typical defaults include the IPv4 source and destination addresses, the protocol number, and the layer 4 source and destination ports. Some platforms add the incoming interface index or a per-device seed so that two routers in series do not make correlated choices, which is called hash polarization.
3. Load balancing also happens below layer 3, on link aggregation groups. A bundle of physical links presented as one logical interface hashes frames across its members, often on MAC and IP fields.
4. That distinction matters for reading names. An interface called `ae66-0` or `be23` is a bundle. Traffic is already being spread across the physical members of that bundle, invisibly, without any change to the hop count.
5. So the reader's trace shows two layers of load balancing at once: ECMP across distinct routers, which is visible as extra addresses, and link bundling inside each hop, which is completely invisible.
6. Classic traceroute increments the UDP destination port per probe, mutating the five-tuple and therefore the hash, which is the anomaly Paris traceroute was written to eliminate.
7. Paris traceroute keeps the five-tuple constant and varies the UDP checksum field instead, adjusting payload bytes so the checksum remains valid. Routers do not hash on the checksum, so all probes follow one path.
8. The reader's hop 11 is the classic textbook case of the anomaly Paris traceroute describes. Reading hops 10, 11 and 12 as a single chain gives a route that probably never existed as one route.

Reader's hop	Route A	Route B
10	ibr02.pnq21	ibr02.pnq21
11	rwa03.pnq21	ibr01.pnq20
12	rwa04.pnq20	rwa04.pnq20

9. **Proven** by this data: at hop 11 the probes reached at least two different routers in two different Pune sites.
10. **Inferred, not proven**: that these form two clean parallel chains as shown. A single-run traceroute cannot separate the chains. Paris traceroute with several fixed flow identifiers, or a multipath detection run that enumerates every parallel path, would settle it.

27.5.6 WORDS - remember these

1. ECMP — using several equally good roads at once — equal-cost multi-path forwarding across multiple next hops of identical routing metric.
2. Five-tuple — the five fields that identify a conversation — source address, destination address, protocol, source port, destination port.
3. Per-flow load balancing — keep each conversation on one road — hashing the flow identifier to select a next hop, preserving packet order.
4. Per-packet load balancing — alternate every packet — round-robin next-hop selection, giving even utilization at the cost of reordering.
5. Link aggregation — several cables acting as one — LAG or bundle interfaces, hashing frames across members below the routing layer.
6. Hash polarization — two routers making the same bad choice — correlated next-hop selection when devices in series use identical hash inputs.

27.6 Stars, and how to read them honestly

27.6.1 PLAIN - in simple words

1. A star in a traceroute means one thing only: no reply arrived for that probe within the waiting time.
2. It does not mean the packet stopped there. It does not mean a router is down. It does not mean anything is broken.
3. There are five common reasons for a star, and only one of them is a fault.
4. Reason one. The router is configured not to send these messages at all. Some operators disable them on purpose.
5. Reason two. The router sends them, but only a few per second, and yours was over the limit. This is very common and it is the default on many systems.
6. Reason three. Something between you and the router blocks the reply on the way back, or blocks your probe on the way out.
7. Reason four. A tunnel technology carries your packet across several machines while showing you only one hop.
8. Reason five. The destination itself is silent by policy, which is normal for large web services.
9. Stars in the middle of a trace, with normal hops after them, always mean a quiet router and never mean a fault. The packets clearly got through.
10. Stars at the end of a trace are the hardest to read, and they are extremely common. Most traces of most large services end in stars.

27.6.2 PLAIN - a picture in your head

1. Go back to the corridor of doors and the paper darts.
2. You throw a dart marked 13 and no card comes back.
3. Six things could have happened, and from where you stand they look identical.
4. Nobody was at position 13, so the dart flew on and nobody kept it.
5. Someone was there but has orders never to walk cards back to strangers.
6. Someone was there, would normally reply, but has already walked one card this second and refuses to walk another.
7. Someone was there and wrote a card, but a doorman between you tore it up.
8. There was a covered walkway between position 12 and position 18, so positions 13 to 17 do not exist as far as darts are concerned.
9. Or the corridor really does end in a wall at position 13.
10. You cannot tell which. Standing at your end and staring harder does not help. You need a different experiment.

Where this comparison breaks: in the corridor, silence is unusual. On the internet, silence is the normal, expected, default state of most routers near a large service. Treat a silent tail as the baseline, not as a surprise.

27.6.3 PLAIN - a worked example

1. The reader's trace ends like this.

```
12  ae106-0.rwa04.pnq20.ntwk.msn.net (104.44.20.42)
13  * * *
14  * * *
15  * * *
16  * * *
17  * * *
18  * * *
19  * * *
20  * * *
```

2. What is proven: no ICMP Time Exceeded message reached the reader's machine for any probe with a hop counter of 13 or more, within the wait time.
3. What is not proven: that the packets stopped at hop 13. That is a guess, and there is no evidence for it in this trace.
4. Compare with a control. Run the same trace to a target known to answer, on the same connection, at the same moment.
5. If that trace also ends in eight stars, the stars belong to the reader's own network or to the general environment, not to this destination.
6. If it completes cleanly, the stars are specific to this path, which narrows things a great deal without proving anything about the cause.
7. The reader did not run that control. It is the single cheapest improvement available, and section 27.13 shows how.

27.6.4 PLAIN - what is really happening inside

1. Generating a Time Exceeded message is expensive for a router, in a way that forwarding is not.
2. Forwarding happens in dedicated hardware: a chip that looks up an address and moves the packet, millions of times a second, without a general-purpose processor involved.
3. Building an error message is different work. The router must copy part of the dead packet, build a new packet around it, work out a source address, and send it. That is software, on the router's ordinary processor.
4. So every serious router limits how much of it can be spent on courtesy messages to strangers. Not out of malice. Out of survival.
5. On Linux the default limit is one such message per second to any given destination, and carrier gear is usually tighter still.
6. Now count. Traceroute sends three probes to the same hop, back to back, in a few milliseconds. Under a one-per-second limit, two of them get nothing.
7. This is why partial stars, like one answer and two stars on the same line, are so common. It is a rate limit, almost every time.
8. Tunnels are the other big cause. If an operator carries your packet inside a wrapper across five internal routers, and the wrapper is set up not to copy your hop counter, those five routers never see your counter reach zero. They vanish from the trace entirely.

27.6.5 TECHNICAL - the engineer's version

1. Linux rate limiting is exposed as sysctls, with documented defaults:

Sysctl	Default	Meaning
`icmp_ratelimit`	1000	ms between replies
`icmp_msgs_per_sec`	1000	global cap per second
`icmp_msgs_burst`	50	burst allowance
`icmp_ratemask`	6168	which types are limited

2. The mask value 6168 is binary with bits 3, 4, 11 and 12 set, meaning ICMP types 3, 4, 11 and 12. Type 11 is Time Exceeded. So Time Exceeded is rate limited by default on Linux, at one per second per destination.
3. RFC 1812 section 4.3.2.8 requires it: a router MUST implement rate limiting of ICMP error messages to avoid being used in a denial-of-service attack. Silence is standards-compliant behaviour, not misconfiguration.
4. Carrier platforms add control-plane policing, a hard rate limit on traffic punted to the route processor, usually classed so that routing protocol packets are protected and ICMP generation is not.

5. MPLS is the main cause of hidden hops. RFC 3032, "MPLS Label Stack Encoding", January 2001, by Rosen and others, specifies that when an IP packet is first labelled, the label's TTL is set from the IP TTL, and on the final pop the IP TTL is set from the label TTL.
6. Operators frequently disable that copying, so the label TTL starts at 255 instead of inheriting yours. Then the whole tunnel decrements a counter your probes never see, and the tunnel interior is invisible.
7. This is an implementation detail with different names per vendor: on Cisco IOS it is `no mpls ip propagate-ttl`, on Junos it is `no-propagate-ttl` under the MPLS protocol stanza.
8. RFC 4950, "ICMP Extensions for Multiprotocol Label Switching", August 2007, by Bonica, Gan, Tappan and Pignataro, allows a label switching router to attach its label stack to ICMP Time Exceeded and Destination Unreachable messages, so a traceroute can display MPLS labels when the operator allows it.
9. Honest ranking of causes of a silent tail, most likely first, for a trace toward a large cloud provider:

Rank	Cause	How common
1	Policy: no ICMP errors	very common
2	Rate limiting	very common
3	Destination silent	very common
4	Filtered on return	common
5	MPLS hiding hops	common
6	Actual black hole	uncommon

10. Note that the genuine fault is last on that list. That ordering is the whole lesson of this section.
11. For the reader's hops 13 to 20: the evidence is consistent with every one of those six causes, and distinguishes between none of them.

27.6.6 WORDS - remember these

1. Star — no reply arrived in time — a probe timeout, recorded as an asterisk, carrying no information about the cause.
2. Rate limiting — only a few replies per second — an enforced minimum interval between generated ICMP messages, required by RFC 1812 4.3.2.8.
3. Control plane — the router's thinking part — the route processor running protocols and generating ICMP, distinct from the forwarding hardware.
4. Control-plane policing — protecting the thinking part — a hard rate limit on traffic punted from the data plane to the route processor.
5. MPLS — a tunnelling technology used inside carriers — Multiprotocol Label Switching, which can hide interior hops by not propagating the IP TTL.
6. Black hole — a place where packets vanish silently — a forwarding or filter state that discards traffic without generating any ICMP notification.

27.7 Why the times look strange

27.7.1 PLAIN - in simple words

1. A normal traceroute prints three times on each line, in milliseconds. One for each probe.
2. People expect those numbers to grow steadily down the page, because each hop is further away. Often they do not.
3. You will regularly see hop 9 answer faster than hop 8. That looks like the packet went backwards in time.
4. It did not. Each number is a full round trip: out to that router, and all the way back to you. It is not the time for that one step of the journey.

5. And the way back can be completely different from the way out. You never see the way back.
6. There is a second cause, and it is bigger than most people expect. Some routers are simply slow at producing the reply, even though they forward traffic at full speed.
7. A router that takes 30 milliseconds to write a complaint will show a high number, while the packets it forwards are delayed by almost nothing.
8. So a slow-looking hop in the middle of a trace, with fast hops after it, is proof that the router is slow at complaining and nothing else.
9. The rule to remember: you cannot subtract two lines to get the time of the link between them.

27.7.2 PLAIN - a picture in your head

1. Imagine measuring the distance to towns along a road by sending a runner to each one with a message, and timing until a reply comes back to you.
2. The reply might come back by a different road, or by train, or by post. Your stopwatch includes all of that.
3. Also, each town has one clerk who writes the replies, and clerks vary. One town's clerk answers in seconds. The next town's clerk finishes lunch first.
4. Now try to work out the distance between town four and town five by subtracting your two stopwatch readings.
5. The answer is meaningless. You have subtracted two different return journeys and two different clerks.

Where this comparison breaks: the outward journey really is shared, so times do usually trend upward over many hops, and a jump that persists across every later hop is real evidence of a long link. The caution is against reading small differences, not the overall shape.

27.7.3 PLAIN - a worked example

1. First, an honest statement about the evidence. The reader's captured trace contains no times at all. It records only hop numbers and addresses.
2. So the example here is a typical trace of the same shape, to show what the reader would have seen.

```
 7  104.44.196.187   18.412 ms  17.998 ms  18.203 ms
 8  104.44.55.163    41.902 ms  42.118 ms  41.777 ms
 9  104.44.53.119    88.640 ms  39.115 ms  39.402 ms
10  104.44.31.62     42.881 ms  43.006 ms  42.744 ms
11  104.44.20.52     42.560 ms  44.881 ms  43.019 ms
12  104.44.20.42     43.702 ms  43.244 ms  43.610 ms
```

3. Read hop 9. One probe took 88 ms and the other two took 39 ms. That is not a congested link. If it were, all three would be slow.
4. It is one reply that queued behind other work on that router's processor. The forwarding path was fine, as hops 10, 11 and 12 prove by being fast.
5. Read hop 10. It is 0.7 ms faster than hop 9, despite being further away. That is well within noise and means nothing.
6. Read the jump between hop 7 and hop 8: 18 ms to 42 ms, and it sticks for every later hop. That is real. Something long sits between them.
7. That is the correct way to use these numbers. Look for jumps that persist, ignore single slow probes, and never subtract.

27.7.4 PLAIN - what is really happening inside

1. Break one printed number into its parts. It contains four things and you can separate none of them.
2. Part one: the time for your probe to travel from you to that router.
3. Part two: the time that router took to notice the dead packet, hand it to its processor, and build the reply. This can be microseconds or tens of milliseconds.

4. Part three: the time for the reply to travel back to you, along a path you cannot see and which need not be the reverse of the outward path.
5. Now consider why the return path is invisible. Internet routing is chosen independently by each network for traffic entering it. The road from A to B and the road from B to A are separate decisions made by different people.
6. Paths being different in the two directions is called asymmetry, and it is the normal case between large networks, not the exception.
7. So when hop 8 shows 42 ms and hop 9 shows 39 ms, the honest reading is: hop 9's reply happened to come home by a quicker road, or hop 9's processor was less busy. Both are ordinary.

27.7.5 TECHNICAL - the engineer's version

1. The measured value is the interval between the probe's transmission timestamp and the reception timestamp of the matching ICMP response, taken in user space by the traceroute process.
2. That value decomposes as forward path delay, ICMP generation latency, and return path delay, plus scheduling jitter on the measuring host itself.
3. ICMP generation latency is the dominant unexplained term. Time Exceeded generation is a control-plane function. On platforms with hardware forwarding, the probe is punted from the data plane to the route processor, queued behind protocol traffic, and answered at low priority.
4. Observed generation latencies of 10 to 100 ms on a busy carrier router are routine while the same router forwards transit traffic with under 100 microseconds of added delay.
5. This is why an intermediate hop can show a far higher RTT than hops beyond it. The high value describes the router's own responsiveness, not the path.
6. Path asymmetry means the return leg is chosen by BGP policy in the far network and may traverse entirely different transit providers, exchange points and even continents.
7. Consequently, per-link latency cannot be derived by differencing adjacent hop RTTs. The difference mixes two different return paths and two different generation latencies, and can be negative.
8. Useful practice for reading times, in order:

Signal	Meaning
One slow probe of three	Control-plane blip
All three slow, one hop	That router is busy
Step up, all later hops	Real long link
Rising loss to the end	Silent policy, not loss

9. Critical caution when reading `mtr` loss columns: loss at an intermediate hop that does not appear at later hops is rate limiting, not packet loss. Only loss that continues to the final hop is real.

27.7.6 WORDS - remember these

1. Round-trip time — there and back — the interval from probe transmission to matching response reception, including generation latency.
2. Path asymmetry — the way back is not the way out — independent forward and reverse routing decisions between autonomous systems.
3. Data plane — the fast part that moves packets — hardware forwarding, typically an ASIC, operating at line rate.
4. Punt — hand a packet to the slow part — transferring a packet from the forwarding hardware to the route processor for software handling.
5. Jitter — variation in delay — the statistical spread of one-way or round-trip latency across repeated measurements.

27.8 What a traceroute cannot tell you

27.8.1 PLAIN - in simple words

1. This is the most important section in the chapter, so read it twice.
2. A traceroute is a weak instrument. It answers one narrow question, and people routinely use it to answer five questions it cannot touch.
3. It cannot tell you the way back. You see only the outward direction.
4. It cannot tell you whether the destination is working. Silence from a destination is normal for almost every large service.
5. It cannot tell you where a block is. The last hop that answered is the last hop that was willing to answer, which is a different thing.
6. It cannot tell you anything at all about what lies past the last reply. Not whether there are more routers, not whether your packets got there.
7. It cannot prove that a hop showing high times is causing your problem.
8. It cannot prove the path is the path your real traffic takes, because your real traffic does not change its port number on every packet.
9. Now say this plainly, because it is the reader's own case.
10. The reader's traceroute, on its own, would NOT have proved a fault. Not the stars, not the twelve hops, not any part of it.
11. Everything in that trace is compatible with a completely healthy network in which GitHub simply does not answer diagnostic probes.
12. Something else proved the fault. The trace only narrowed the region.

27.8.2 PLAIN - a picture in your head

1. A traceroute is a torch shone down a corridor, at night, from one end.
2. It lights up the people who choose to wave back. It does not light up the corridor.
3. Someone standing still and silent is invisible. So is a wall. So is an open door with nothing beyond it.
4. Now imagine you conclude "the corridor ends at position 13" because nobody waves past 12.
5. You have confused "nobody waved" with "nothing is there". That is the error almost everyone makes with traceroute.

Where this comparison breaks: the torch is worse than it sounds, because it also does not show you the route the wave took to come back. Two people standing next to each other can wave back along completely different routes, and their apparent distance from you will differ for that reason alone.

27.8.3 PLAIN - a worked example

1. Here is the reader's situation, split into what was proven and what was merely suggested.

Observation	What it proves
12 hops answered	Those 12 forwarded and replied
Hops 13-20 silent	Nothing on its own
Path enters Microsoft	Handoff happened in Delhi
Names resolve	Reverse DNS exists

2. Now the observations that came from outside the traceroute, in the same session, and what those proved.

Observation	What it proves
curl: no reply in 15 s	No TCP answer at all
No RST, no ICMP error	Silent drop, not refusal
Works on mobile data	Server is up and serving
DNS resolved fine	Not a name problem

3. Compare the two tables. The second one carries almost all of the weight.
4. The decisive fact is the combination: the same request to the same address succeeded over a different network, at the same time.
5. That rules out the destination being down, because it served the phone.
6. It also rules out DNS, because both used the same address.
7. What remains is that something on the path used by the home connection discarded the traffic without saying so.
8. The traceroute contributed one thing to that conclusion: it showed which networks were involved, so the problem could be described precisely instead of vaguely.

27.8.4 PLAIN - what is really happening inside

1. The deep reason traceroute is weak is that it measures the wrong traffic.
2. Its probes are not your traffic. They have different ports, often a different protocol, a different size, and a deliberately broken hop counter.
3. Every device on the path is entitled to treat them differently from your real traffic, and many do.
4. So a perfect traceroute does not prove your real connection will work, and a completely silent traceroute does not prove it will fail.
5. There is a second, subtler weakness. Traceroute reports on the forward direction only, but a connection needs both directions to work.
6. A failure in which your packets arrive perfectly and the replies are discarded looks, from your end, exactly like a failure in which your packets never arrive.
7. Traceroute cannot separate those two cases. Nothing you run from one end can. You need a measurement from the other end.
8. This is why the honest diagnostic sequence is: use traceroute to describe the region, then use a test with real traffic to establish the fault, then get a second viewpoint from elsewhere to locate it.

27.8.5 TECHNICAL - the engineer's version

1. Enumerated limits of a single-ended forward traceroute:

Cannot determine	Why
Reverse path	Never observed
Destination liveness	Silence is policy
Filter location	Only shows repliers
State past last reply	No data at all
Real traffic's path	Different flow ID
Which direction failed	One-ended view

2. The probe traffic differs from production traffic in transport, port, payload size, packet rate and TTL, so per-hop treatment can differ under access control lists, QoS classification, policy-based routing and stateful inspection.
3. The last responding hop is the last hop willing and able to generate an ICMP error that reached you. It carries no information about topology beyond itself.

4. A common and serious error is naming the operator of the last responding hop as the cause. In the reader's case that would mean blaming Microsoft's Pune router, which the data does not support in any way.
5. Correct statements for the reader's evidence, split by strength:
6. **Proven.** Twelve hops responded, ending at `104.44.20.42`, a Microsoft address inside `104.40.0.0/13`, announced by AS8075. No response arrived for TTL values 13 through 20.
7. **Proven, from outside the trace.** A TCP connection to `20.207.73.82` port 443 produced no response of any kind in 15 seconds, and the identical request over a mobile network succeeded immediately.
8. **Supported, not proven.** Traffic on this path was silently discarded somewhere at or beyond the last responding hop.
9. **Not supported by any evidence here.** That a specific named party blocked it, that the discard point is hop 13, or that the trace itself demonstrates a fault.

27.8.6 WORDS - remember these

1. Single-ended measurement — testing from one side only — a measurement with no observation point in the far network, blind to the reverse path.
2. Last responding hop — the last router that answered — the furthest TTL for which an ICMP error was received, with no topological meaning beyond that.
3. Silent drop — discarded without notice — a discard action that generates no ICMP error, defeating all ICMP-based diagnosis.
4. Liveness — whether something is actually working — reachability and service availability, which ICMP silence cannot establish either way.
5. Looking glass — a window into someone else's network — a public read-only interface for running ping, traceroute and BGP queries from an operator's own routers.

27.9 The reader's trace, decoded hop by hop

27.9.1 PLAIN - in simple words

1. The trace has four zones, and once you see them the whole thing reads easily.
2. Zone one is hop 1. That is the box in the reader's own flat.
3. Zone two is hops 2 to 6. That is the reader's internet provider, using addresses that are not visible from the outside world.
4. Inside zone two, hop 3 stands out. It has an address of the kind used on the public internet, sitting between two hops that do not.
5. Zone three is hops 7 to 12. Every one of those belongs to Microsoft, which owns GitHub, and the names say which city each is in.
6. Zone four is hops 13 to 20, which is silence.
7. So the journey visible in this trace is: home, provider, Microsoft, quiet.

27.9.2 PLAIN - a picture in your head

1. Think of posting a parcel and watching the tracking page fill in.
2. First scan: your own doorstep.
3. Next few scans: your local courier's vans and depots, with internal codes that mean nothing to anyone outside that company.
4. Then a scan at a handover point, where the parcel changes companies.
5. Then several scans inside the second company, whose codes are different and happen to include city names you recognize.
6. Then the tracking page stops updating.

Where this comparison breaks: a tracking page stops updating because nobody scanned the parcel, and that is exactly right here too. But a parcel is one object with one history. This trace is twenty different parcels, and some of the scans belong to journeys that never met.

27.9.3 PLAIN - a worked example

1. The whole trace in one table. Names are shown without the shared ending `.ntwk.msn.net`, which every one of them carries.

Hop	Address	Name	What it is
1	192.168.0.1	none	reader's own router
2	172.31.0.17	none	ISP access, private
3	137.97.29.249	none	ISP, public address
4	172.26.22.235	none	ISP core, private
5a	172.16.18.33	none	ISP core, private
5b	172.26.14.75	none	ISP core, private
5c	172.26.22.234	none	ISP core, private
6a	172.26.14.75	none	same as hop 5b
6b	172.16.18.33	none	same as hop 5a
7	104.44.196.187	ae66-0.del01-96cbe-1b	Microsoft, Delhi
8a	104.44.55.163	be23.rwa02.bom01	Microsoft, Mumbai
8b	104.44.51.55	po22.rwa04.bom01	Microsoft, Mumbai
9a	104.44.53.119	be1010.owr02.bom01	Microsoft, Mumbai
9b	104.44.53.47	be1013.owr01.bom01	Microsoft, Mumbai
10	104.44.31.62	be5.ibr02.pnq21	Microsoft, Pune
11a	104.44.20.52	ae104-0.rwa03.pnq21	Microsoft, Pune
11b	104.44.16.51	be1.ibr01.pnq20	Microsoft, Pune
12	104.44.20.42	ae106-0.rwa04.pnq20	Microsoft, Pune
13-20	none	none	no reply at all

2. Rows marked `a`, `b` and `c` are different probes at the same hop count. They are not separate hops.

27.9.4 PLAIN - what is really happening inside

1. **Hop 1, `192.168.0.1`.** The reader's own router, in the flat. It is the default gateway: the machine every packet leaving the home goes to first. The address is in `192.168.0.0/16`, a range reserved for private use, so millions of homes use this exact number at the same time.
2. **Hop 2, `172.31.0.17`.** The first machine inside the provider. Still a private address, from the `172.16.0.0/12` range, which covers everything from `172.16.x.x` to `172.31.x.x`. This is the provider's access network.

3. **Hop 3, `137.97.29.249`.** A public address, and the only one before Microsoft. It belongs to the reader's provider. Its presence here tells you the provider mixes public and private numbering inside its own core.
4. **Hop 4, `172.26.22.235`.** Back to private. This is not a mistake and not a loop. Providers commonly number their internal links privately even when the same routers also hold public addresses on other interfaces.
5. **Hop 5, three addresses.** `172.16.18.33`, `172.26.14.75` and `172.26.22.234`. Three probes, three parallel links, three routers. Note that `.234` is one below hop 4's `.235`, which is what you expect from two interfaces on the same small link subnet.
6. **Hop 6, two addresses.** `172.26.14.75` and `172.16.18.33`. Both appeared at hop 5. The same routers are five hops away by one road and six by another, so they answer at both distances.
7. **Hop 7, `104.44.196.187`.** The handover. The name ends in `ntwk.msn.net`, which is Microsoft's backbone naming domain, and the site code is `del01`, meaning Delhi. This is where the provider hands the traffic to Microsoft.
8. **Hop 8, two Mumbai routers.** `104.44.55.163` and `104.44.51.55`, named `rwa02` and `rwa04` at site `bom01`. Mumbai. Two parallel paths, both inside Microsoft.
9. **Hop 9, two more Mumbai routers.** `104.44.53.119` and `104.44.53.47`, named `owr02` and `owr01`. Still Mumbai, one layer deeper in.
10. **Hop 10, `104.44.31.62`.** Named `ibr02.pnq21`. Pune. Both parallel paths converge here, because only one address appeared.
11. **Hop 11, two Pune routers in two buildings.** `104.44.20.52` at site `pnq21` and `104.44.16.51` at site `pnq20`. The paths diverge again, and the two roads are not the same length.
12. **Hop 12, `104.44.20.42`.** Named `rwa04.pnq20`. Pune. The last machine on the whole internet that said anything at all to the reader's computer.

27.9.5 TECHNICAL – the engineer's version

1. Address classification, with the governing document:

Address	Block	Status
192.168.0.1	192.168.0.0/16	private, RFC 1918
172.31.0.17	172.16.0.0/12	private, RFC 1918
137.97.29.249	public	reader's ISP
172.26.x, 172.16.x	172.16.0.0/12	private, RFC 1918
104.44.x.x	104.40.0.0/13	Microsoft, AS8075

2. RFC 1918, "Address Allocation for Private Internets", February 1996, reserves `10.0.0.0/8`, `172.16.0.0/12` and `192.168.0.0/16`. The `/12` spans `172.16.0.0` to `172.31.255.255`, so all of hops 2, 4, 5 and 6 fall inside it.
3. **Proven** for the Microsoft hops: `104.44.196.187` lies inside `104.40.0.0/13`, a prefix announced by AS8075, registered to Microsoft Corporation. Every hop from 7 to 12 shares that block.
4. GitHub was acquired by Microsoft in 2018, and `github.com` in this session resolved to `20.207.73.82`, inside `20.192.0.0/10`, also AS8075. So the Microsoft hops are not a detour; they are the destination network.
5. On hop 4 and hop 5c: `172.26.22.235` and `172.26.22.234` differ by one. Two readings are possible and a traceroute cannot separate them.
6. Reading one: they are the two ends of a single point-to-point link, addressed as a `/31` under RFC 3021, "Using 31-Bit Prefixes on IPv4 Point-to-Point Links", December 2000, which gives exactly two usable addresses, or as a `/30` giving two usable out of four.
7. Reading two: they are unrelated interfaces that merely sit close together in the same `/24` the operator uses for core links.

8. Mark this clearly as **inferred**. Nothing in a traceroute reveals a prefix length.
9. Hop 3 being public while hops 2, 4, 5 and 6 are private is worth stating carefully. It does not imply the packet left the ISP at hop 3 and came back. A single router can hold public addresses on some interfaces and private addresses on others, and the ICMP source is the interface the reply left by.
10. Reverse DNS: the names on hops 7 to 12 come from PTR lookups performed by the reader's own resolver, `1.1.1.1`, after the trace collected the addresses. Hops 1 to 6 show no names because the ISP publishes no PTR records for that private space, which is normal and correct.
11. Verify any of this yourself with `whois 104.44.196.187`, `dig -x 104.44.196.187 @1.1.1.1`, or an RDAP query against the ARIN registry.

27.9.6 WORDS - remember these

1. Default gateway — the router your machine sends everything to — the next hop for destinations outside the local subnet.
2. Private address — an address only meaningful inside one network — a block from RFC 1918, never routed on the public internet.
3. Reverse DNS — turning an address back into a name — a PTR record lookup under `in-addr.arpa`, published at the address holder's discretion.
4. Autonomous system — one network under one routing policy — an AS such as AS8075 for Microsoft, identified by number in BGP.
5. Point-to-point link — a cable with exactly two ends — a link addressed with a `/30` or, under RFC 3021, a `/31`.

27.10 Reading carrier router names

27.10.1 PLAIN - in simple words

1. Large network operators give every router a name, and the name is built from parts that describe the machine.
2. The parts usually say: which cable the packet came in on, which building the router is in, which city that is, and what job the router does.
3. Once you can read them, a trace stops being a wall of characters and becomes a sentence about geography.
4. Important warning first. There is no standard for this. No specification says how to name a router.
5. Every operator invents its own scheme, and schemes change over time inside the same company.
6. So everything in this section is a **convention**: a pattern people follow because it is useful, not a rule anyone is obliged to obey.
7. Read names as strong hints, never as proof. If a name says Mumbai and the times say otherwise, believe the times.

27.10.2 PLAIN - a picture in your head

1. Think of a hospital room label: `2W-14B-ICU`.
2. Floor 2, west wing, room 14, bed B, intensive care.
3. Nobody outside that hospital could invent that code, but anyone inside reads it instantly and knows exactly where to walk.
4. Another hospital across the road uses a completely different code for the same idea, and both are perfectly sensible.
5. Router names work exactly like that, and `ntwk.msn.net` is one hospital's labelling system.

Where this comparison breaks: a hospital label is put there by the hospital for its own use, and so is a router name, but you are reading it from outside with no key. Some parts are guessable and some, like a device serial code, are not. Say which is which rather than inventing meanings for the rest.

27.10.3 PLAIN - a worked example

1. Take the reader's hop 7 and split it at the dots.

```
ae66-0 . del01-96cbe-1b . ntwk . msn . net
  |            |            |     |     |
  |            |            |     |     +-- top-level domain
  |            |            |     +-------- Microsoft's domain
  |            |            +-------------- network naming zone
  |            +--------------------------- site and device
  +---------------------------------------- interface on that device
```

2. Read it right to left, the way domain names are actually structured.
3. `net` is the top-level domain. `msn` is a Microsoft domain, in use since the Microsoft Network launched in 1995. `ntwk` is short for network, the zone Microsoft uses for its backbone infrastructure.
4. `del01` is the site: Delhi, facility 01. `96cbe-1b` is a device identifier that is not publicly documented, so do not guess at it.
5. `ae66-0` is the interface: aggregated Ethernet bundle 66, logical unit 0.
6. Putting it in a sentence: "logical unit 0 of bundle 66 on a device at Microsoft's Delhi site 1".

27.10.4 PLAIN - what is really happening inside

1. The name did not travel with the packet. The router never told anyone its name.
2. What happened is that traceroute collected an address, then asked DNS to turn that address back into a name. That is a reverse lookup.
3. The operator publishes those names in DNS voluntarily, one per interface address, usually generated automatically from its own inventory system.
4. That is why the interface appears in the name at all. There is one name per interface address, not one name per router.
5. So a single router shows up under different names at different hops, depending on which door your probe came in through.
6. It also means the names can be stale. An automatic system that is not run after a change leaves an old name in place for months.
7. And it means the names can be absent, which is what happened on the reader's hops 1 to 6.

27.10.5 TECHNICAL - the engineer's version

1. City codes in `ntwk.msn.net` follow IATA airport codes, the three-letter codes used on luggage tags. The three in the reader's trace, verified:

Code	City	Airport
del	New Delhi	DEL, Indira Gandhi
bom	Mumbai	BOM, Chhatrapati Shivaji
pnq	Pune	PNQ, Pune Airport

2. The trailing digits are facility numbers within the metro, so `pnq20` and `pnq21` are two distinct Pune sites, not two routers in one building.
3. Role codes, as used by Microsoft. These are a **convention** of one operator, not an industry standard, and other carriers use entirely different letters:

Code	Conventional meaning
ibr	inter-region backbone router
rwa	regional wide-area router

Code	Conventional meaning
owr	outer wide-area router

4. Reading the roles as a shape: `ibr` devices carry traffic between regions over long-haul links, `rwa` devices aggregate within a region, and `owr` devices sit further out toward peers and customers.
5. Interface prefixes all mean the same underlying thing, several physical links bonded into one logical link, but the word differs by vendor:

Prefix	Full name	Vendor usage
ae	aggregated Ethernet	Juniper Junos
be	bundle-Ether	Cisco IOS-XR
po	port-channel	Cisco IOS, NX-OS

6. Seeing `ae`, `be` and `po` in one trace is normal and suggests a multi-vendor network, which every large operator runs deliberately.
7. The `-0` in `ae66-0` and `ae104-0` is the logical unit or subinterface number, Junos style. `be23` and `po22` carry no unit, which is the Cisco style of writing the same idea.
8. Full decode of every hostname in the reader's trace. All nine end in `.ntwk.msn.net`, which is omitted here for width.

Hostname	Interface	Site and role
ae66-0.del01-96cbe-1b	ae 66, unit 0	Delhi 01, edge device
be23.rwa02.bom01	bundle 23	Mumbai 01, rwa 02
po22.rwa04.bom01	port-channel 22	Mumbai 01, rwa 04
be1010.owr02.bom01	bundle 1010	Mumbai 01, owr 02
be1013.owr01.bom01	bundle 1013	Mumbai 01, owr 01
be5.ibr02.pnq21	bundle 5	Pune 21, ibr 02
ae104-0.rwa03.pnq21	ae 104, unit 0	Pune 21, rwa 03
be1.ibr01.pnq20	bundle 1	Pune 20, ibr 01
ae106-0.rwa04.pnq20	ae 106, unit 0	Pune 20, rwa 04

9. Hop 7 is the odd one out: it has no three-letter role code, only `del01-96cbe-1b`. **Inferred**, not proven: this is an edge or peering device named from an inventory identifier rather than a backbone role code, which is consistent with it being the handover point from the reader's ISP.
10. Honest limits of name reading, in order of importance: names are published voluntarily and may be absent; they may be stale after a change; the codes are one operator's private scheme; and geolocation databases built on such names inherit every one of those errors.
11. Cross-check a name before relying on it. Compare the round-trip time with the distance the name implies. A hop named Mumbai answering in 2 ms from Delhi is a stale or wrong name, not a physics breakthrough.

27.10.6 WORDS - remember these

1. IATA code — the three-letter airport code — the International Air Transport Association location identifier, widely reused for network site names.
2. Site code — which building the router is in — the operator's facility identifier, usually a city code plus a number.
3. Role code — what job the router does — a naming convention indicating the device's place in the network hierarchy, private to each operator.

4. Bundle interface — several cables acting as one — a link aggregation group, called `ae` on Junos, `Bundle-Ether` on IOS-XR, `Port-channel` on IOS.
5. Subinterface — a logical slice of one interface — a numbered unit on a physical or bundle interface, often carrying one VLAN.

27.11 The trace as a story

27.11.1 PLAIN - in simple words

1. Read the whole thing as one journey and it becomes memorable.
2. A packet leaves a laptop in a flat and reaches a plastic box on a shelf.
3. The box passes it to the provider's access network, the equipment that serves the street and the neighbourhood.
4. It crosses the provider's own core, several machines using addresses that only mean something inside that company.
5. It reaches the provider's public edge, the point where that company touches the rest of the internet.
6. It is handed to Microsoft in Delhi.
7. It travels down the country to Mumbai, then inland to Pune.
8. And then nothing more is heard from it.

27.11.2 PLAIN - a picture in your head

1. Picture a parcel leaving a house and reaching the corner post box.
2. A van takes it to the local depot, then a bigger van to the regional hub.
3. At the hub it is handed to a different company entirely, with different vans and different paperwork.
4. That company flies it from Delhi to Mumbai, then drives it to Pune.
5. At Pune the tracking page stops. The parcel is somewhere in that building or somewhere beyond it, and nobody will say which.

Where this comparison breaks: a parcel is one object, and this is not. Twenty probes went out and twelve zones answered. Also, no van driver in this story was ever obliged to scan anything for you, and most of them did it as a courtesy that they can withdraw at any time.

27.11.3 PLAIN - a worked example

1. The journey in one diagram.

```
flat  ->  home router  ->  ISP access  ->  ISP core
[1]       192.168.0.1      172.31.0.17     172.26.x
                                           172.16.x
                                              |
                                              v
                                        ISP public edge
                                        137.97.29.249
                                              |
                                              v
  Microsoft DELHI  ->  MUMBAI  ->  PUNE  ->  silence
  104.44.196.187       104.44.5x     104.44.2x   * * *
     hop 7             hops 8, 9     hops 10-12  13 to 20
```

2. Now the distances. These are straight-line figures and are approximate.

Leg	Straight km	Fastest possible RTT
Flat to ISP edge	under 50	under 1 ms
Delhi to Mumbai	about 1,150	about 11.5 ms

Leg	Straight km	Fastest possible RTT
Mumbai to Pune	about 120	about 1.2 ms

3. Where the fastest possible figure comes from: light in glass fibre travels at roughly 200,000 km per second, which is about 5 microseconds per kilometre one way, so 10 microseconds per kilometre there and back.
4. Real cable never runs straight, and real routers add delay, so expect real figures well above the floor.

Leg	Floor	Typical real RTT
Home to ISP edge	under 1 ms	2 to 15 ms
Delhi to Mumbai	11.5 ms	25 to 40 ms
Mumbai to Pune	1.2 ms	3 to 8 ms

5. The typical figures are approximate working numbers for Indian domestic routes, not measurements from this session.

27.11.4 PLAIN - what is really happening inside

1. Notice what the story reveals about the reader's provider: it did not hand the traffic to Microsoft locally. It carried it to Delhi first.
2. Notice also that Microsoft carried it a long way inside its own network, from Delhi to Pune, roughly 1,180 km in a straight line.
3. That is normal for a large content network. Once traffic is inside, moving it on private long-haul fibre is cheap and fully controlled.
4. **Inferred, not proven**: that the reader is physically near Delhi. The Delhi handover is equally consistent with a provider that backhauls traffic from elsewhere to a Delhi interconnection point.
5. What is **proven** is only the sequence of sites: the handover happened at a Delhi-named Microsoft device, and the last reply came from a Pune-named one.
6. The destination address itself, `20.207.73.82`, is also a Microsoft address, so the trace was heading toward the same network the whole time from hop 7 onward.

27.11.5 TECHNICAL - the engineer's version

1. The handoff at hop 7 is an interconnection: either a private interconnect or a public internet exchange port, between the reader's ISP and AS8075.
2. A traceroute cannot distinguish those two. Both appear as one hop with the far side's address.
3. Once inside AS8075, forwarding is Microsoft's own decision. The Delhi to Mumbai to Pune ordering reflects the internal topology, and the `ibr` role code at hops 10 and 11 is consistent with inter-region long-haul devices.
4. Rough latency budget for the whole visible path, using floor values plus realistic equipment and routing overhead:

Segment	Expected RTT
Hops 1 to 3	2 to 15 ms
Hop 3 to hop 7	5 to 20 ms
Delhi to Mumbai	25 to 40 ms
Mumbai to Pune	3 to 8 ms

5. Total expected round trip at hop 12 is therefore roughly 40 to 80 ms, which is unremarkable for a domestic Indian path to a cloud edge.

6. This estimate cannot be checked against the reader's capture, because that capture contains no timing data. Note that as a gap in the evidence rather than filling it in.

27.11.6 WORDS - remember these

1. Access network — the equipment serving your street — the last-mile aggregation between customer premises and the provider core.
2. Interconnection — where two networks touch — a private interconnect or internet exchange port between two autonomous systems.
3. Backhaul — carrying traffic to a bigger site before handing it off — long transport from access aggregation to a core or peering location.
4. Long-haul — a link between distant cities — high-capacity inter-city fibre, typically terrestrial DWDM within a country.
5. Latency floor — the fastest the distance allows — propagation delay at roughly 200,000 km per second in fibre, about 5 microseconds per kilometre.

27.12 What the silence after hop 12 means

27.12.1 PLAIN - in simple words

1. Eight hop numbers, twenty-four probes, not one answer.
2. There are five plausible explanations, and they are not equally likely.
3. One. Microsoft's inner network does not send these diagnostic replies to outsiders, by policy. Very likely.
4. Two. Those routers do send them, but only a few per second, and traceroute asked too fast. Very likely.
5. Three. The destination itself never answers this kind of probe. Very likely, and true of almost every large web service.
6. Four. Something on the way discarded the reader's packets silently. Possible.
7. Five. Something discarded the replies on the way back. Possible.
8. The first three are ordinary behaviour that would look exactly the same on a perfectly working connection.
9. So the honest conclusion from the trace alone is: no conclusion.

27.12.2 PLAIN - a picture in your head

1. You knock on twenty doors down a corridor. Twelve open. Eight do not.
2. Behind those eight doors could be an empty room, a person who ignores knocks, a person already busy answering someone else, or a wall.
3. You will not learn which by knocking harder.
4. But you have learned something real: the corridor definitely continues to door twelve, and door twelve is in Pune.

Where this comparison breaks: knocking on a real door has one meaning. A traceroute probe carries a hop counter, and a device can forward it perfectly while refusing to answer, so a silent door here may be one your packets are sailing straight past.

27.12.3 PLAIN - a worked example

1. Ranked honestly, most likely first, for a trace toward a large cloud provider's inner network.

Rank	Explanation	Fault?
1	Policy: no ICMP errors	no
2	ICMP rate limiting	no

Rank	Explanation	Fault?
3	Destination stays silent	no
4	Return replies filtered	no
5	MPLS hides interior hops	no
6	Genuine silent discard	yes

2. Five of the six most likely explanations are not faults.
3. And note that explanation 6, the only real fault, is indistinguishable from the other five using this tool.
4. What the trace did contribute: it proved the traffic reached Microsoft's network and travelled inside it as far as a Pune device.
5. That narrows the region enormously. It rules out the reader's own router, the reader's provider, and any question of a wrong address.
6. It does not identify the cause. Not even slightly.

27.12.4 PLAIN - what is really happening inside

1. The reason silence is uninformative is that all six causes produce the same observation at the reader's end: no packet arrives.
2. To distinguish them you need an observation that cannot be produced by politeness, only by a working or a broken path.
3. That is what the `curl` test provided. It did not ask for a courtesy reply. It attempted a real connection to a real service on port 443.
4. A working path produces an answer to that within milliseconds. A refusing server produces a refusal. A missing route produces an error message.
5. The reader got none of those. Fifteen seconds, and nothing of any kind.
6. Then the same request, to the same address, over mobile data, succeeded immediately.
7. That pair of results is what settled it, and Chapter 28 works through exactly why.
8. Keep the two roles separate in your head: the traceroute describes the region, and the connection test establishes the fault.

27.12.5 TECHNICAL - the engineer's version

1. **Proven.** No ICMP Time Exceeded arrived for TTL 13 through 20 within the configured wait, across 24 probes.
2. **Not proven, and not supported by this data alone.** That forwarding stops at hop 13; that any specific party filtered anything; that the destination is unreachable.
3. Discriminating experiments, from cheapest to most conclusive:

Test	What it separates
Same trace, TCP 443	Filtering by probe type
Trace to a live control	Local versus path-specific
`mtr` for 100 cycles	Rate limiting from real loss
`curl` to port 443	Politeness from real failure
Reverse trace inbound	Which direction fails

4. The rate-limiting hypothesis is testable directly: rerun with `-q 1` and a pause between probes. If hops appear that were stars at `-q 3`, the cause was rate limiting, not filtering.
5. The MPLS hypothesis is testable by looking for a discontinuity: a sudden large RTT step across one hop boundary, or ICMP extensions carrying label information as specified in RFC 4950.

6. The direction question cannot be answered from one end at all. It requires a probe originated inside or near the destination network, which is what a looking glass or a remote measurement platform provides.
7. Final honest reading of the reader's evidence, in one sentence: the traffic entered Microsoft's network in Delhi and was last seen alive at a Pune device, after which it was discarded silently somewhere, by something, in a direction that this trace cannot determine.

27.12.6 WORDS - remember these

1. Discriminating test — an experiment whose outcomes differ by cause — a test designed so that competing hypotheses predict different observations.
2. Silent discard — dropping without notification — a filter or forwarding action producing no ICMP error, common on security policy denies.
3. Narrowing — reducing the region under suspicion — eliminating candidate causes without identifying the actual one.
4. Control test — the same test against a known-good target — a baseline run that separates environment-specific from target-specific behaviour.
5. Bidirectional visibility — seeing both directions — measurement from both ends, required to determine which direction of a path fails.

27.13 How to run a much better traceroute

27.13.1 PLAIN - in simple words

1. The default traceroute is a 1987 tool with 1987 defaults, and it is the wrong tool for a modern web problem.
2. Four changes make it far better, and all four are one-letter flags.
3. Change the probe type to match the traffic that is actually failing. For a website, that means a real web connection to port 443.
4. Turn off name lookups. During a network fault, name lookups are slow or broken and they make the tool look stuck when it is not.
5. Shorten the waiting time, so a silent path finishes in seconds instead of minutes.
6. Then run the same command again against something known to work, so you have a comparison.
7. After that, run a tool that keeps probing, because one snapshot cannot show a fault that comes and goes.
8. Finally, get somebody else to trace back toward you, because you cannot see the return path from where you are standing.

27.13.2 PLAIN - a picture in your head

1. A doctor who takes one temperature reading, of one patient, with a broken thermometer, has learned very little.
2. Take the reading with the right instrument. Take it again on a healthy person to check the instrument. Take it repeatedly over time. And ask a colleague at the other hospital what they see.
3. Those four steps are exactly the four commands below.

Where this comparison breaks: a patient stays still. A network path changes between measurements, so two runs a minute apart can legitimately differ, and that difference is itself information rather than an error.

27.13.3 PLAIN - a worked example

1. The commands, for macOS, in the order to run them.

```
# 1. The real test: TCP to the port that is failing.
sudo traceroute -P TCP -p 443 -n -q 1 -w 2 -m 30 20.207.73.82

# 2. The control: identical flags, a target known to answer.
sudo traceroute -P TCP -p 443 -n -q 1 -w 2 -m 30 1.1.1.1

# 3. For comparison, the ICMP method Windows uses by default.
sudo traceroute -I -n -q 3 -w 2 -m 30 20.207.73.82

# 4. The classic UDP method, which needs no special permission.
traceroute -n -q 3 -w 2 -m 30 20.207.73.82

# 5. Continuous, with per-hop loss, over TCP port 443.
sudo mtr -T -P 443 -n --report --report-cycles 100 20.207.73.82
```

2. What each flag does.

Flag	Meaning
`-P TCP`	Send TCP probes, not UDP
`-p 443`	Aim at the HTTPS port
`-I`	Send ICMP echo instead
`-n`	Do not look up names
`-q 1`	One probe per hop
`-w 2`	Wait 2 seconds, not 5
`-m 30`	Stop after 30 hops

3. Why `-q 1` with TCP: each TCP probe leaves a half-open connection at the target until it times out. Sending fewer is both faster and politer.
4. Why `-n` matters most during a fault: without it, every hop triggers a reverse name lookup, and if DNS is affected the tool appears to hang.
5. Worst case with these flags is 30 hops times 1 probe times 2 seconds, which is 60 seconds. With the defaults it is 450 seconds.

27.13.4 PLAIN - what is really happening inside

1. A **looking glass** is a web page or login service run by a network operator that lets outsiders run a few read-only commands on that operator's own routers.
2. Typically you can run ping, traceroute, and queries about which routes that operator knows.
3. Operators provide them so that other engineers can diagnose problems involving their network without having to phone anybody.
4. Why it matters here: a looking glass inside or near the destination network can trace back toward the reader's address.
5. That gives you the return path, which is the one thing a traceroute from your own machine can never show you.
6. Where to find them: most large operators publish one, and there are public directories of them maintained by the peering community. Measurement platforms such as RIPE Atlas do the same job from thousands of probes.
7. A cheaper version of the same idea, which the reader already had available: run the test from a phone on mobile data. That is a second viewpoint on a completely different path.

27.13.5 TECHNICAL - the engineer's version

1. Flag reference, quoting the macOS manual page:

Flag	Documented meaning
`-P proto`	UDP, TCP, GRE and ICMP
`-I`	ICMP ECHO, synonym for -P icmp
`-p port`	Base port, default 33434
`-q n`	Probes per ttl, default 3
`-w s`	Wait, default 5 sec
`-m n`	Max ttl, default from sysctl
`-n`	Numeric addresses only
`-f n`	Set the first ttl used

2. `-P TCP` and `-I` require raw sockets, so run them with `sudo`.
3. `mtr` flags used above: `-T` selects TCP probes, `-P 443` sets the port, `-n` disables name lookup, `--report` prints a summary rather than a live display, `--report-cycles 100` sets how many rounds to run.
4. Reading `mtr` output correctly is the single highest-value skill here: loss at an intermediate hop that does not persist to the final hop is ICMP rate limiting, not packet loss. Only loss that continues to the last hop is real.
5. For flow-stable path discovery use `paris-traceroute`, or `scamper` from CAIDA in its multipath detection mode, or `dublin-traceroute`. These map the parallel paths instead of blending them.
6. For a reproducible record, save output with a timestamp, for example by redirecting to a file named with the date, and keep the control run alongside it. A trace without a control run is much weaker evidence.
7. Recommended minimum evidence set for reporting a path fault to a provider: the TCP traceroute to the failing target, the identical traceroute to a working control, an `mtr` report of at least 100 cycles, the `curl -v` output showing the failure, and the same `curl` from a second network.

27.13.6 WORDS - remember these

1. Looking glass — a public window onto an operator's routers — a read-only interface offering ping, traceroute and BGP route queries.
2. Control run — the same test on something known to work — a baseline measurement isolating target-specific behaviour from environmental noise.
3. Reverse traceroute — a trace run toward you from elsewhere — a measurement originated near the destination, revealing the return path.
4. RIPE Atlas — a global network of small measurement probes — a distributed active measurement platform operated by the RIPE NCC.
5. Half-open connection — a connection request never completed — a TCP session left in SYN-RECEIVED at the target, consuming state until it times out.

27.98 Common wrong ideas

1. Wrong: traceroute asks the network to report the path. Right: there is no such request. It provokes error messages by sending packets designed to die.
2. Wrong: the trace shows the route your traffic takes. Right: it shows a blend of routes taken by dozens of separate probes, each of which may be hashed onto a different link.
3. Wrong: TTL means time, so a slow link uses more of it. Right: it counts hops. One router costs exactly one, whether it holds the packet for a microsecond or a second.
4. Wrong: stars mean the packet stopped there. Right: stars mean no reply arrived. The packet may have sailed straight through a router that simply does not answer.

5. Wrong: several addresses on one hop is a bug. Right: it is load balancing across parallel links, and it is a sign of a well-built network.
6. Wrong: subtract two hops' times to get the latency of the link between them. Right: each time is a full round trip on a different return path, with different reply-generation delays. The subtraction is meaningless.
7. Wrong: a slow hop in the middle is causing your problem. Right: if the hops after it are fast, that router is merely slow at generating replies, and its forwarding is fine.
8. Wrong: the last responding hop is where the fault is. Right: it is the last hop that was willing and able to answer, which says nothing about what lies beyond it.
9. Wrong: the reader's traceroute proved a fault. Right: it proved nothing of the kind. Every line of it is compatible with a healthy network. The `curl` timeout plus success on mobile data is what established the fault.
10. Wrong: router hostnames are a standard you can rely on. Right: they are each operator's private convention, published voluntarily, sometimes stale, and always to be cross-checked against measured times.

27.99 Chapter summary in 20 lines

1. IP has no field that records a path, so there is no way to ask for one.
2. Traceroute works by abusing the TTL field, a safety counter meant only to stop packets looping forever.
3. TTL is 8 bits, counts hops not seconds, and every forwarding router must decrement it by exactly one, per RFC 1812 section 5.3.1.
4. Common starting values are 64 on Linux and macOS, 128 on Windows and 255 on much network gear, which lets you guess a sender's system, weakly.
5. Send TTL 1 and the first router discards the packet and returns ICMP Time Exceeded, type 11 code 0, sourced from its own interface address.
6. Repeat with TTL 2, 3, 4 and so on. Three probes per hop is the default, and each round-trip time is measured per probe.
7. Van Jacobson wrote traceroute in 1987 at Lawrence Berkeley Laboratory, from a suggestion by Steve Deering.
8. The classic method sends UDP to ports from 33434 upward; Windows `tracert` sends ICMP echo; TCP traceroute sends a SYN to a real port such as 443.
9. TCP to 443 gets furthest through modern filtering because it looks exactly like the traffic the path is built to carry. It is the right tool here.
10. Multiple addresses on one hop are equal-cost multi-path load balancing. The reader's hops 5, 6, 8, 9 and 11 all show it, and all are normal.
11. Classic traceroute changes its destination port per probe, which changes the load-balancing hash. Paris traceroute, from 2006, fixes exactly that.
12. Stars mean only that no reply arrived. Causes include policy, rate limiting, filtering, MPLS hiding hops, and a silent destination.
13. Linux rate limits ICMP Time Exceeded to one per second per target by default, which alone explains most partial star lines.
14. Stars at the end of a trace are extremely common and prove nothing. The reader's hops 13 to 20 are consistent with six different causes.
15. Round-trip times can go down as you go further out, because they include reply-generation time on a slow control plane and an unseen return path.
16. You cannot subtract two hops' times to get a link latency.
17. The reader's trace reads: flat, home router, ISP access, ISP core with one public address at hop 3, Microsoft's Delhi edge, Mumbai, Pune, silence.
18. Names decode as `ntwk.msn.net` for Microsoft's backbone, IATA city codes del, bom and pnq for Delhi, Mumbai and Pune, role codes ibr, rwa and owr, and interface prefixes ae, be and po all meaning bonded link bundles.

19. Say it plainly: the reader's traceroute alone would not have proved a fault. It narrowed the region and nothing more.
20. What settled it was a real connection attempt that received no response of any kind, and the same request succeeding over mobile data.

28

CHAPTER

KB

TCP, Ports, and Why a Timeout Is Evidence

28.0 What this chapter gives you

1. You will be able to say exactly what the internet's basic delivery service promises, and the much longer list of things it does not promise.
2. You will be able to explain TCP and UDP as two answers to the same problem, and say why UDP is not a worse TCP but a different tool.
3. You will be able to explain a port as a 16-bit number, and a socket as the four numbers that make one conversation unique on the whole planet.
4. You will be able to draw the three-way handshake with real sequence numbers, walk the eleven states of a connection, and explain the four-way close.
5. You will be able to explain how TCP turns an unreliable service into a reliable one, and what it costs in round trips.
6. You will be able to separate flow control from congestion control, and say why a link that loses one packet in a hundred destroys throughput.
7. You will be able to name the three possible endings of a connection attempt, and tell them apart from the failure message alone.
8. You will be able to state precisely what the reader's fifteen-second silence proves, and be equally precise about what it does not prove.
9. You will be able to run `curl -v`, `nc -vz`, `nmap` and `tcpdump` and say what each one adds to the evidence.

28.1 What IP alone gives you, and what it does not

28.1.1 PLAIN – in simple words

1. The internet's basic delivery service is called IP, short for Internet Protocol. It moves one small parcel of data from one address to another.
2. That parcel is called a packet. It carries a from-address, a to-address, and some data.
3. IP makes exactly one promise, and it is a weak one: it will try. The official phrase for it is **best effort**.
4. IP does not promise the packet will arrive, nor that ten packets sent in order will arrive in order, nor that a packet arrives only once.
5. Worst of all, IP does not promise to tell you when a packet is lost. Nobody rings a bell. The packet just stops existing.
6. So something had to be built on top of IP that turns "try" into "it arrived, complete, in order, exactly once". That something is TCP.

28.1.2 PLAIN – a picture in your head

1. Imagine a postal system with no tracking, no receipts and no complaints desk.
2. Most postcards arrive. Some do not. Nobody tells you which.

3. Now send a 300-page book through this system, one page per postcard. The receiver gets a pile with pages missing, out of order, maybe one twice.
4. To fix it you would number every page, and the receiver would write back saying which numbers arrived, and you would re-post the gaps.
5. That numbering-and-writing-back scheme, done automatically, is TCP.

Where this comparison breaks: real post loses letters mostly by accident. IP loses packets mostly on purpose. A router with a full queue drops the newest packet deliberately, because dropping is the only signal it has to tell senders to slow down. Loss is not a malfunction in IP. Loss is the brake pedal.

28.1.3 PLAIN - a worked example

1. Suppose you send ten packets numbered 1 to 10 across the internet.
2. A router in the middle has a full queue for a few milliseconds and drops two.
3. Two others take a different path because the network is load balancing, the same behaviour the reader saw at hops 5, 6, 8, 9 and 11 of their traceroute.

Sent	Arrived	Order seen	Note
1 and 2	yes	1st, 2nd	fine
3	no	-	dropped in queue
4	yes	4th	out of order
5	yes	3rd	took faster path
6	no	-	dropped in queue
7 to 10	yes	5th to 8th	fine

4. The receiver has eight packets, two gaps, and one pair swapped. IP has done its job correctly. Nothing is broken.
5. Nobody sent any message saying "3 and 6 are gone". The sender must work it out by noticing that 3 and 6 were never confirmed.
6. That is precisely what TCP does, and it is the whole reason TCP exists.

28.1.4 PLAIN - what is really happening inside

1. Each router looks only at the destination address in the packet header, decides the next hop, and forwards it.
2. Routers hold packets in a fixed-size queue while the outgoing link is busy. When the queue is full, an arriving packet is discarded. That rule is called tail drop, and no record of the packet is kept anywhere.
3. Sometimes the router sends back a complaint using ICMP, the Internet Control Message Protocol, the mechanism behind "host unreachable" and `traceroute`.
4. But ICMP is carried in IP packets, so complaints can be lost too, and many networks throttle or block ICMP on purpose.
5. The reader saw exactly that: hops 13 to 20 of their traceroute returned `* * *`, which usually means ICMP replies were suppressed, not a broken path.

28.1.5 TECHNICAL - the engineer's version

1. IPv4 is RFC 791, published September 1981 by Jon Postel at the Information Sciences Institute for DARPA. IPv6 is RFC 8200, July 2017.
2. The IPv4 service model is unreliable, connectionless, best-effort datagram delivery. There is no connection state in any router.
3. The idea comes from Vint Cerf and Bob Kahn's paper "A Protocol for Packet Network Intercommunication", IEEE Transactions on Communications, May 1974.

4. In 1978 the design was deliberately split into IP for delivery and TCP for reliability, by Cerf, Postel and Danny Cohen, so packet voice could skip the reliability part. That split is why UDP exists. ARPANET switched from NCP to TCP/IP on 1 January 1983, a cutover still remembered as "flag day".

Property	Provided by IP	Provided by TCP
Addressing of hosts	yes	no
Delivery guarantee	no	yes
Ordering	no	yes
Duplicate removal	no	yes
Data integrity check	header only	full segment
Identify the program	no	yes, via ports

5. Observe raw IP behaviour with `ping -s` for size effects, `traceroute` for the path, and `tcpdump -n -v ip` to read headers on the wire.

28.1.6 WORDS - remember these

1. Packet — one small parcel of data with an address on it — an IP datagram carrying a header and a payload.
2. Best effort — we will try, we promise nothing — a service model with no delivery, ordering or duplicate-suppression guarantees.
3. ICMP — the network's complaint messages — the Internet Control Message Protocol, RFC 792, advisory and frequently filtered.

28.2 TCP and UDP: two ways to use IP

28.2.1 PLAIN - in simple words

1. Sitting directly on top of IP there are two main choices, and almost everything you use picks one of them.
2. **TCP**, the Transmission Control Protocol, is the careful one. It sets up a connection first, numbers everything, checks everything arrived, and puts it back in order.
3. **UDP**, the User Datagram Protocol, is the plain one. It adds almost nothing. It sends your parcel and forgets about it.
4. It is tempting to say TCP is good and UDP is bad. That is wrong. TCP's care costs time, and repairing a missing piece can be worse than living without it.
5. In a live voice call, a lost slice of sound from 300 milliseconds ago is worthless, so a voice call uses UDP and carries on. A file download uses TCP, because a file with a hole in it is not a file.
6. TCP trades time for completeness. UDP trades completeness for time.
7. A third name you will hear is QUIC. QUIC runs on top of UDP and rebuilds reliability in a smarter way. Chapter 32 covers it properly.

28.2.2 PLAIN - a picture in your head

1. TCP is a registered parcel service with signatures. Before sending, the courier phones ahead and confirms someone is home.
2. Every parcel is numbered and the receiver signs for each one. If a signature does not come back, the courier sends that parcel again.
3. UDP is a person throwing newspapers from a bicycle. No phone call, no signature, no numbering, no second attempt. Fast, cheap, mostly fine.
4. For a newspaper, mostly fine is fine. For a contract it is a disaster.

Where this comparison breaks: the courier does not really phone ahead to a human. The three-way handshake is between two pieces of software, and it exists mainly to agree on starting numbers, not to check that anyone is home. Also, the newspaper thrower does at least check the paper is not torn: UDP has a checksum, optional over IPv4 and mandatory over IPv6.

28.2.3 PLAIN – a worked example

1. Here are the two headers side by side, drawn to the same scale. Each row is 32 bits, which is 4 bytes.

```
TCP header, 20 bytes minimum, up to 60 with options

 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-------------------------------+-------------------------------+
|          Source Port          |       Destination Port        |
+-------------------------------+-------------------------------+
|                        Sequence Number                        |
+---------------------------------------------------------------+
|                    Acknowledgement Number                     |
+-------+-----------+-----------+-------------------------------+
| Offset|  Reserved |   Flags   |            Window             |
+-------+-----------+-----------+-------------------------------+
|           Checksum            |        Urgent Pointer         |
+-------------------------------+-------------------------------+
|                    Options, 0 to 40 bytes                     |
+---------------------------------------------------------------+
```

```
UDP header, exactly 8 bytes, always

+-------------------------------+-------------------------------+
|          Source Port          |       Destination Port        |
+-------------------------------+-------------------------------+
|            Length             |           Checksum            |
+-------------------------------+-------------------------------+
```

2. UDP then stops. TCP keeps going for another 12 bytes minimum, and those 12 bytes are the entire machinery of reliability.
3. Sending 100 bytes by UDP costs 8 bytes of overhead. By TCP it costs at least 20 bytes plus a three-packet handshake: a bad deal for one short message, invisible for a 50 MB download.

28.2.4 PLAIN – what is really happening inside

1. When a program sends by UDP, the operating system writes an 8-byte header, hands the result to IP, and the call returns. Nothing is remembered, and if the message is lost, no software anywhere notices.
2. When a program sends by TCP, the operating system copies the data into a send buffer belonging to that connection, and it sits there until confirmed.
3. The kernel keeps a timer per connection. If a confirmation does not arrive in time, it sends the data again from the buffer. That is why TCP needs a connection: it is the place to keep the buffer, the timers and the numbering.
4. And TCP does not preserve your write boundaries. Three writes of 100 bytes may arrive as one read of 300. TCP is a stream of bytes, not messages.
5. UDP does preserve boundaries. One send equals one receive, or nothing at all. That single difference decides many design arguments.

28.2.5 TECHNICAL – the engineer's version

1. TCP is now specified in RFC 9293, "Transmission Control Protocol (TCP)", STD 7, published August 2022, edited by Wesley Eddy.

2. RFC 9293 obsoletes the original RFC 793 of September 1981 and folds in RFC 879, RFC 2873, RFC 6093, RFC 6429, RFC 6528 and RFC 6691. If you cite one document for TCP today, cite RFC 9293.
3. UDP is RFC 768, published 28 August 1980 by Jon Postel. It is three pages long. That is the entire specification and it has never needed replacing.
4. QUIC is RFC 9000, May 2021. It runs over UDP and implements its own loss recovery, congestion control and encryption. HTTP/3 over QUIC is RFC 9114, June 2022. Chapter 32 covers this in full.

Application	Transport	Why
HTTP/1.1 and HTTP/2	TCP	needs every byte
HTTP/3	QUIC over UDP	faster setup
SSH, git over SSH	TCP	needs every byte
Email, SMTP, IMAP	TCP	needs every byte
DNS query	UDP mostly	one small round trip
DNS over TLS	TCP	size and privacy
Voice and video call	UDP	late data is useless
Live game state	UDP	newest state wins
NTP time sync	UDP	tiny, retry is cheap
DHCP address lease	UDP	no address yet
Streaming video	TCP or QUIC	buffered, needs all bytes

5. Note also that streaming video is not a UDP application. Netflix and YouTube send video over TCP or QUIC, because a video file with holes will not decode. It is live conversation, not video, that needs UDP.
6. Observe both with `ss -tan` and `ss -uan` on Linux, or `netstat -an -p tcp` on macOS.

28.2.6 WORDS - remember these

1. TCP — the careful delivery service — a connection-oriented, reliable, ordered byte-stream transport protocol, RFC 9293.
2. UDP — the plain delivery service — a connectionless, unreliable, message-oriented transport protocol, RFC 768.
3. QUIC — a modern replacement — a UDP-based transport with integrated TLS 1.3 and per-stream loss recovery, RFC 9000.

28.3 Ports, sockets, and the four-tuple

28.3.1 PLAIN - in simple words

1. An IP address gets a packet to the right machine. It does not say which program on that machine should receive it, and one laptop may run a browser, a mail client and a chat app all talking to the network at once.
2. So each packet also carries a number saying which program it is for. That number is called a **port**.
3. A port is a 16-bit number. Sixteen bits gives 65,536 values, 0 to 65,535.
4. Every TCP packet carries two: a source port, saying which program sent it, and a destination port, saying which program should get it.
5. Servers use fixed, agreed numbers so clients know where to knock. Web servers using TLS listen on port 443. Everyone knows this.
6. Clients need no fixed number. When your browser opens a connection, the operating system hands it a spare high-numbered port, used once and released.

28.3.2 PLAIN - a picture in your head

1. Think of a large office block with one street address, which is the IP address. It gets the post van to the building.
2. Inside there are 65,536 numbered pigeonholes. A pigeonhole number is a port.
3. Reception sorts every letter into a pigeonhole by the number on it. Some pigeonholes have a standing arrangement: 443 is always the web desk, 22 is always the remote-access desk, and anyone can look this up.

Where this comparison breaks: a real pigeonhole holds one item at a time. Port 443 on a busy server holds hundreds of thousands of simultaneous conversations, because a conversation is identified by four numbers together, not by the port alone. The next block fixes that picture properly.

28.3.3 PLAIN - a worked example

1. When the reader ran `curl -v https://github.com`, the operating system picked a spare local port. On macOS these come from the range 49152 to 65535.
2. Say it picked 51514. The connection is then described by four numbers.

```
source IP        192.168.0.101      the laptop, on the home LAN
source port      51514              picked by macOS, used once
destination IP   20.207.73.82       what DNS returned for github.com
destination port 443                the agreed port for HTTPS
```

3. Those four values together are the **four-tuple**, and they identify this one conversation and no other.
4. The laptop address here is an example inside the reader's home range: the session record kept the router address `192.168.0.1` but not the laptop's own. Everything else is exactly as recorded.

Field	Tab 1	Tab 2	Someone else
Source IP	192.168.0.101	192.168.0.101	203.0.113.9
Source port	51514	51515	51514
Dest IP	20.207.73.82	20.207.73.82	20.207.73.82
Dest port	443	443	443

5. Look at column two and column four. Same source port, 51514, and no conflict, because the source IP differs. The tuple as a whole must be unique. That is how one server port holds hundreds of thousands of conversations at once.

28.3.4 PLAIN - what is really happening inside

1. Inside the operating system there are two kinds of socket, and confusing them causes real bugs.
2. A **listening socket** is a server saying "send me anything arriving on port 443". It is identified by only two things: a local address and a local port. It never carries data; its only job is to catch new connection attempts.
3. A **connected socket** is one live conversation, identified by all four values: local IP, local port, remote IP, remote port.
4. When a connection request arrives, the kernel finds the listening socket, creates a brand new connected socket for that one client, and queues it.
5. The server program calls `accept`, which takes one connected socket off the queue and hands it over as a new file descriptor.
6. When a packet arrives, the kernel first matches all four values against connected sockets. Only if nothing matches does it try listening sockets.
7. If nothing matches at all, the kernel replies with a TCP RST, which produces the message "Connection refused". Remember that reply. It is the centre of this whole chapter.

28.3.5 TECHNICAL - the engineer's version

1. Port numbers are 16-bit unsigned integers. IANA divides the space by RFC 6335, published August 2011.
2. System Ports are 0 to 1023 and on Unix require root or a capability to bind. User Ports are 1024 to 49151. Dynamic or Private Ports are 49152 to 65535.
3. Ephemeral port ranges are an implementation detail, not a standard. Linux defaults to 32768 to 60999 in `net.ipv4.ip_local_port_range`. macOS and the BSDs default to 49152 to 65535 via `net.inet.ip.portrange.first`.

Port	Proto	Service	What it is for
20	TCP	FTP data	file bytes, old
21	TCP	FTP control	commands only
22	TCP	SSH	shell, scp, git over SSH
23	TCP	Telnet	plain text, obsolete
25	TCP	SMTP relay	server to server mail
53	UDP+TCP	DNS	name to address
67	UDP	DHCP server	hands out leases
68	UDP	DHCP client	receives the lease
80	TCP	HTTP	web, no encryption
110	TCP	POP3	fetch and delete mail
123	UDP	NTP	clock synchronization
143	TCP	IMAP	mail kept on server
443	TCP+UDP	HTTPS	web over TLS, QUIC on UDP

Port	Proto	Service	What it is for
465	TCP	SMTPS submission	mail send, implicit TLS
587	TCP	SMTP submission	mail send, STARTTLS
853	TCP	DNS over TLS	private DNS queries
993	TCP	IMAPS	IMAP with implicit TLS
3306	TCP	MySQL, MariaDB	database
3389	TCP	RDP	Windows remote desktop
5432	TCP	PostgreSQL	database
6379	TCP	Redis	in-memory data store
8080	TCP	HTTP alternate	dev servers, proxies

4. Port 443 carries HTTP over TLS, defined for that use in RFC 2818, May 2000. Port 853 for DNS over TLS is RFC 7858, May 2016. Port 465 was revoked and then restored for submission with implicit TLS by RFC 8314, January 2018; port 587 for submission is RFC 6409.
5. Port 443 became the universal port because firewalls and captive networks nearly always permit outbound 443, and because the traffic is encrypted, so middleboxes cannot easily inspect or rewrite it.
6. So HTTPS is now a general-purpose tunnel. Git, container registries, remote desktops, VPNs and messaging protocols all offer a 443 mode. That is a convention driven by firewall reality, not a design goal.
7. It is directly relevant to the reader's session: `git` over SSH uses port 22, `git` over HTTPS uses port 443, and in the reader's outage those two transports did not behave the same way.
8. Inspect with `ss -ltn` on Linux, `lsof -i -P -n` on macOS, and `ss -tn state established` for live connections.

28.3.6 WORDS - remember these

1. Port — a number saying which program a packet is for — a 16-bit unsigned field in the TCP or UDP header.
2. Four-tuple — the four numbers that make one conversation unique — source IP, source port, destination IP, destination port.
3. Ephemeral port — a temporary number for outgoing calls — a dynamically assigned local port, 49152 to 65535 on macOS, 32768 to 60999 on Linux.

28.4 The three-way handshake

28.4.1 PLAIN - in simple words

1. Before TCP carries a single byte of your data, the two sides exchange three small packets. This is the **three-way handshake**.
2. Packet one, client to server: I want to open a connection, and my counting starts at this number.
3. Packet two, back: agreed, I heard your number, and my counting starts at this other number. Packet three, forward: agreed, I heard your number too.
4. After those three, both sides know the other is there, both know where the other's numbering starts, and data can flow.
5. The names come from flags in the header. The first packet has the SYN flag set, for synchronize. The second has SYN and ACK. The third has ACK.
6. Notice the cost. The client cannot send data until packet three, and packet three needs packet two first. That is one complete round trip of pure waiting before any useful byte moves.
7. This handshake is also exactly where the reader's problem lived. Their packet one went out and packet two never came back.

28.4.2 PLAIN - a picture in your head

1. Two people want to talk on a bad radio link, and must be sure both directions work before they start.
2. Aisha says: "Aisha here, my messages will be numbered starting at 481."
3. Ben replies: "Ben here, I heard 481, so I expect 482 next. Mine start at 907."
4. After Ben's reply, Aisha knows the link works both ways, because she got an answer. But Ben knows his link works only one way until Aisha's third message arrives. That is why three messages are needed and not two.

Where this comparison breaks: real people can repeat themselves and improvise. TCP cannot. If Ben's reply is lost, Aisha has no way to tell whether Ben never heard her or Ben answered and the answer vanished. She can only send her opening line again and wait. Section 28.8 is about what she may conclude from silence.

28.4.3 PLAIN - a worked example

1. Here is the reader's own connection, packet by packet, with realistic numbers filled in. The starting numbers are chosen unpredictably by each side.

```
laptop 192.168.0.101:51514        server 20.207.73.82:443
   state: CLOSED                     state: LISTEN
        |                                  |
        |  SYN                             |
        |  seq = 1842315776                |
        |  ack = 0     win = 65535         |
        |  options: mss 1460, wscale 6, SACK|
        |--------------------------------->|
   SYN_SENT                            SYN_RECEIVED
        |                                  |
```

```
        |                    SYN, ACK      |
        |                    seq=3091447201|
        |                    ack=1842315777|
        |                    win = 65535   |
        |<----------------------------------|
        |                                   |
        |  ACK                              |
        |  seq = 1842315777                 |
        |  ack = 3091447202  win = 131712   |
        |---------------------------------->|
   ESTABLISHED                          ESTABLISHED
        |                                   |
        |  TLS ClientHello, 517 bytes       |
        |  seq = 1842315777                 |
        |---------------------------------->|
```

2. Read the acknowledgement numbers carefully. The server's ack is 1842315777, the client's start plus one.
3. That plus one is not a data byte. The SYN flag itself consumes one number in the sequence space, so the handshake can be acknowledged reliably. The client's ack of 3091447202 does the same to the server's start.
4. The window value 65535 is a raw 16-bit field. The `wscale 6` option means "multiply my future windows by 2 to the power 6", which is 64. So the later window of 131712 means about 8.4 megabytes of buffer space.

28.4.4 PLAIN – what is really happening inside

1. The application calls `connect`. The kernel allocates a connection control block, picks a source port, picks an initial sequence number, moves the socket to state SYN_SENT, and sends the SYN.
2. A retransmission timer starts, initially one second.
3. At the server, the SYN arrives and the kernel checks whether any socket is listening on port 443. If yes, it creates a small half-open entry, replies with SYN-ACK, and enters SYN_RECEIVED.
4. When the client's ACK arrives, the server moves the entry into the accept queue, marks it ESTABLISHED, and wakes any program waiting in `accept`.
5. On the client, the SYN-ACK cancels the retransmission timer, moves the socket to ESTABLISHED, and unblocks the `connect` call.
6. Only now does the application start writing. For HTTPS the first thing written is a TLS ClientHello, not the web request.
7. If the SYN-ACK never arrives, the timer fires and the SYN is sent again with the wait doubled. That behaviour is the whole of section 28.8.

28.4.5 TECHNICAL – the engineer's version

1. RFC 9293 section 3.5 defines the handshake. The three segments carry these exact control bits.

Segment	Flags set	Seq	Ack
1, client to server	SYN	ISN_c	not valid
2, server to client	SYN, ACK	ISN_s	ISN_c + 1
3, client to server	ACK	ISN_c + 1	ISN_s + 1

2. Initial sequence numbers must not be predictable. RFC 9293 section 3.4.1, folding in RFC 6528 by Fernando Gont and Steven Bellovin, February 2012, recommends this construction.

```
ISN = M + F(localip, localport, remoteip, remoteport, secretkey)
```

```
M  = a timer that ticks about every 4 microseconds
F  = a cryptographic hash, keyed with a host secret
```

3. The reason is security. If an attacker can guess your next sequence number, they can inject data or forge a connection without ever seeing your packets. That is a blind in-window attack.
4. The famous demonstration was the attack on Tsutomu Shimomura's machines on 25 December 1994, attributed to Kevin Mitnick, which used sequence-number prediction with IP source spoofing.
5. Steven Bellovin had described the weakness in his 1989 paper "Security Problems in the TCP/IP Protocol Suite", and later wrote RFC 1948, May 1996, the first randomization recommendation. RFC 5961, August 2010, adds hardening against blind RST and SYN injection.
6. The SYN and FIN flags each consume one sequence number, so they can be acknowledged and retransmitted with the same machinery as data.
7. Handshake cost is exactly one round-trip time before the first application byte. TLS 1.3 adds one more round trip and TLS 1.2 adds two, as Chapter 29 covers.
8. Observe with `tcpdump -n 'tcp[tcpflags] & tcp-syn != 0'`.

28.4.6 WORDS - remember these

1. Handshake — the short exchange that opens a connection — the SYN, SYN-ACK, ACK sequence defined in RFC 9293 section 3.5.
2. SYN — the flag meaning "let us start counting" — the synchronize control bit, which consumes one sequence number.
3. ACK — the flag meaning "I received up to here" — the acknowledgement control bit, making the acknowledgement number field valid.
4. Initial sequence number — the random starting count — the ISN, chosen per RFC 6528 as a timer plus a keyed hash of the four-tuple.

28.5 The TCP state machine and the four-way close

28.5.1 PLAIN - in simple words

1. A TCP connection is not just a pipe. It is a small machine with a memory of where it currently is. That memory is called its **state**.
2. There are eleven states worth knowing, and a connection moves between them only when a specific packet arrives or the program does something.
3. Opening is fast: three packets and you are in ESTABLISHED, the state where data flows.
4. Closing is slower, because a TCP connection is really two one-way streams, and each direction must be closed separately.
5. So closing takes four packets, not three. One side says "I am done sending". The other says "understood". Then the same happens in the other direction.
6. After the last packet, the side that started the close sits in a state called TIME_WAIT for a while, doing nothing. That looks like waste. It is not: it catches late packets from the connection that just ended.

28.5.2 PLAIN - a picture in your head

1. Think of a phone call between two polite people who both hate hanging up first.
2. Aisha says "that is all from me". Ben says "understood, noted". Ben may still talk, and Aisha listens. The call is half-closed.
3. But Aisha does not put the phone down immediately. She holds the line for a couple of minutes, because if her final "understood" was not heard, Ben will repeat his goodbye, and somebody has to answer it.

Where this comparison breaks: real people can tell a repeated goodbye from a new call. A computer reusing the same four numbers cannot, which is the real reason for the wait. It is not politeness, it is

identity. The next connection using the same four-tuple must not receive a stray packet meant for the last one.

28.5.3 PLAIN - a worked example

1. Here is the four-way close. The client closes first, so the client ends up waiting.

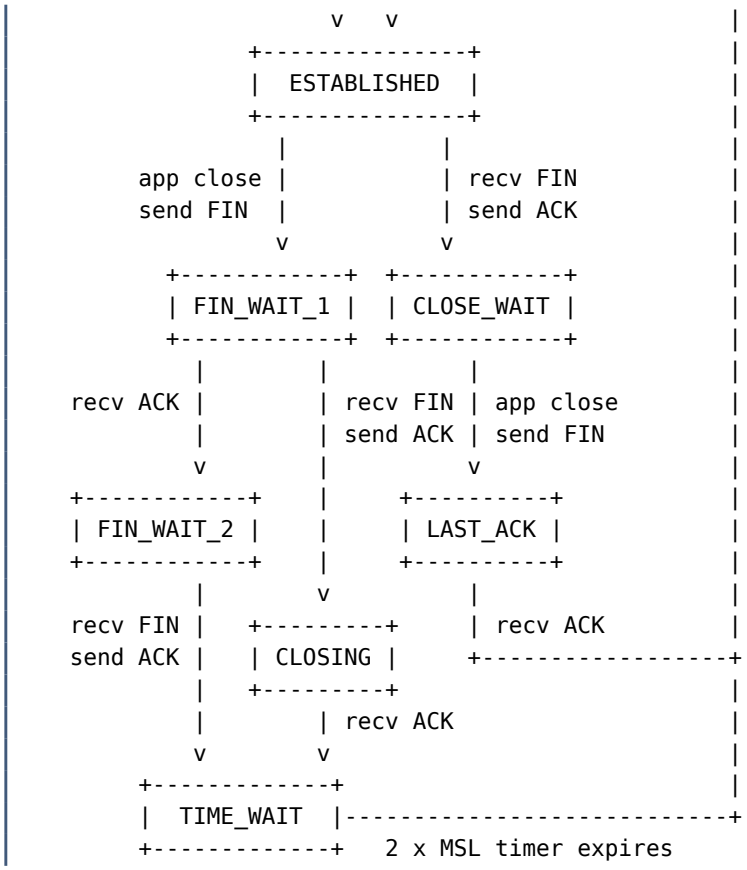
```
client (closes first)              server
  ESTABLISHED                        ESTABLISHED
       |                                  |
       |  FIN, ACK   seq = x              |
       |--------------------------------->|
   FIN_WAIT_1                          CLOSE_WAIT
       |                                  |
       |            ACK   ack = x + 1     |
       |<---------------------------------|
   FIN_WAIT_2                             |
       |     (server may still send data) |
       |            FIN, ACK  seq = y     |
       |<---------------------------------|
   TIME_WAIT                           LAST_ACK
       |                                  |
       |  ACK   ack = y + 1               |
       |--------------------------------->|
       |                                 CLOSED
   wait 2 x MSL
       |
     CLOSED
```

2. Count the packets: FIN, ACK, FIN, ACK. Four. Often the middle two are combined into one FIN-ACK, when the server has nothing left to send.
3. `2 x MSL` means twice the maximum segment lifetime. RFC 9293 sets MSL at 2 minutes, so the specified wait is 4 minutes.
4. Linux uses a fixed 60 seconds instead, compiled in as `TCP_TIMEWAIT_LEN`. That is an implementation detail, not a standard.

28.5.4 PLAIN - what is really happening inside

1. Here is the full state machine. Read it as: you are in a box, an event happens, you move along an arrow.

```
                      +--------+
              +------->| CLOSED |<--------------+
              |        +--------+               |
              |          |      |               |
    app calls listen     |    app calls connect |
    (passive open)       |    send SYN          |
              |          |      |               |
              v          |      v               |
       +--------+        |  +----------+        |
       | LISTEN |        |  | SYN_SENT |        |
       +--------+        |  +----------+        |
            |            |       |              |
     recv SYN            |   recv SYN,ACK       |
     send SYN,ACK        |   send ACK           |
           v             |       |              |
   +--------------+      |       |              |
   | SYN_RECEIVED |------+       |              |
   +--------------+  recv ACK    |              |
           |                     |              |
           +--------+   +--------+              |
```



2. SYN_SENT is a client that has sent its SYN and is waiting. This is exactly where the reader's connection was stuck.
3. SYN_RECEIVED is the server's half-open state between SYN-ACK and the final ACK. ESTABLISHED is the working state where everything useful happens.
4. FIN_WAIT_1 and FIN_WAIT_2 belong to the side that closed first, before and after its FIN is acknowledged. CLOSE_WAIT belongs to the side that was told first, and thousands of CLOSE_WAIT sockets is nearly always an application bug: the program never called `close` itself.

28.5.5 TECHNICAL – the engineer's version

1. The eleven states are normative in RFC 9293 section 3.3.2. The names are the same strings printed by `netstat` and `ss` on every platform.
2. TIME_WAIT exists for two distinct reasons, and people usually remember only the first.
3. Reason one: to let any duplicate or delayed segment from this incarnation of the connection expire before the same four-tuple can be reused.
4. Reason two: to be able to retransmit the final ACK. If that ACK is lost, the peer in LAST_ACK resends its FIN and somebody must answer. Without TIME_WAIT the answer would be an RST, which the peer logs as an error.
5. Only the side performing the active close enters TIME_WAIT. Whoever calls `close` first pays the cost.
6. This is why servers accumulate TIME_WAIT. An HTTP server that closes each connection after responding, or a proxy that opens and closes short-lived back-end connections, is the active closer every time.

Item	Specified	Linux	macOS
MSL	2 minutes	not used directly	`net.inet.tcp.msl`
TIME_WAIT length	2 x MSL = 240 s	fixed 60 s	2 x msl
FIN_WAIT_2 timeout	none	60 s, `tcp_fin_timeout`	sysctl controlled

7. At 60 seconds of TIME_WAIT that caps sustained new connections at roughly 470 per second to a single destination. The fix is connection reuse, not tuning.
8. `net.ipv4.tcp_tw_recycle` was different, was unsafe behind NAT, and was removed from Linux in version 4.12, July 2017. Advice telling you to enable it is out of date and should be ignored.
9. Inspect with `ss -tan state time-wait | wc -l` on Linux, or `netstat -an -p tcp | grep -c TIME_WAIT` on macOS.

28.5.6 WORDS - remember these

1. State — where a connection currently is — one of the eleven values in the RFC 9293 state machine.
2. ESTABLISHED — the working state — both directions open, data may flow.
3. TIME_WAIT — the pause after hanging up — the 2 x MSL wait held by the active closer to absorb stray segments and resend the final ACK.
4. CLOSE_WAIT — waiting for your own program to hang up — the state after receiving a FIN, exited only when the application calls `close`.

28.6 Making TCP reliable and fast

28.6.1 PLAIN - in simple words

1. TCP's reliability rests on one habit: number everything, and confirm what you received.
2. Every byte in the stream has a number. Not every packet, every byte.
3. The receiver keeps saying "I have everything up to number N". That message is the acknowledgement, or ACK.
4. The sender keeps a copy of everything sent but not yet confirmed, and if a confirmation does not arrive in time, sends it again.
5. Doing that one piece at a time would be safe but slow, so the sender is allowed to send many pieces before waiting for any confirmation. The amount allowed in flight at once is called the **window**, and as confirmations come in it slides forward, letting new data out.
6. The receiver controls the window size, so a fast sender cannot drown a slow receiver. That is **flow control**.

28.6.2 PLAIN - a picture in your head

1. Think of a conveyor belt between a warehouse and a shop.
2. The shop tells the warehouse how many crates it has room for. Say twelve. The warehouse may put up to twelve crates on the belt without hearing back.
3. Each crate is numbered. The shop calls out "received everything up to 7", which frees the warehouse to put crates 13 to 19 on the belt.
4. If crate 5 falls off, the shop keeps calling "everything up to 4", no matter how many later crates arrive, and the warehouse works out that 5 is missing.

Where this comparison breaks: the numbering in TCP counts bytes, not crates, so a crate's number is the number of the first item inside it. Also the belt is not a single fixed-speed belt: the sender is guessing how fast the road can take traffic, which is section 28.7, and that guess is separate from the storeroom.

28.6.3 PLAIN - a worked example

1. Suppose the sender's window is 5 segments of 1,460 bytes, starting at sequence number 1000, and segment 3 is lost.

```
sent:  seq 1000  1460 bytes   arrives
       seq 2460  1460 bytes   arrives
       seq 3920  1460 bytes   LOST
       seq 5380  1460 bytes   arrives
       seq 6840  1460 bytes   arrives
```

```
acks:  ack 2460                after first
       ack 3920                after second
       ack 3920   (duplicate)  after fourth
       ack 3920   (duplicate)  after fifth
       ack 3920   (duplicate)  after a sixth segment
```

2. The receiver cannot say "I have 5380", because that would claim everything below it, and 3920 is missing. So it repeats 3920. This is a **cumulative acknowledgement**: all bytes below this number are safely here.
3. The sender sees the same acknowledgement three times over. Three duplicates is the agreed signal.
4. It immediately re-sends just the segment at 3920, without waiting for any timer. That is **fast retransmit**.
5. The extra mechanism is **selective acknowledgement**, SACK. The receiver adds a note saying "I also have the block from 5380 to 8300".
6. With SACK the sender re-sends exactly one segment. Without it, up to three.

28.6.4 PLAIN - what is really happening inside

1. The sender keeps three markers over its send buffer: the oldest unconfirmed byte, the next byte to send, and the end of the window.
2. The waiting time before declaring a loss is measured, not fixed. The sender keeps a smoothed average of the round-trip time and a measure of how much that time jumps around, then sets the timeout to average plus four times the jumpiness.
3. On the receiving side, out-of-order segments are held in a reorder queue and handed to the application only once the gap is filled.
4. If the application is slow to read, that queue fills and the advertised window shrinks. At zero the sender stops sending data entirely.
5. The sender then sends a tiny **window probe** every so often, because if it simply waited for an update and that update were lost, both sides would wait forever.

28.6.5 TECHNICAL - the engineer's version

1. The retransmission timeout is computed per RFC 6298, June 2011, by Paxson, Allman, Chu and Sargent, with alpha = 1/8, beta = 1/4 and K = 4.

```
first measurement R:
    SRTT   = R
    RTTVAR = R / 2

later measurements R':
    RTTVAR = (1 - beta) * RTTVAR + beta * |SRTT - R'|
    SRTT   = (1 - alpha) * SRTT  + alpha * R'
    RTO    = SRTT + max(G, K * RTTVAR)

then clamp:  RTO = max(RTO, 1 second)
```

2. Here is that running on a path of about 30 ms with one spike.

Sample	RTT ms	SRTT ms	RTO used
1	30.0	30.00	1000 ms, floored
2	32.0	30.25	1000 ms, floored
3	28.0	29.97	1000 ms, floored
4	95.0	38.10	1000 ms, floored
5	31.0	37.21	1000 ms, floored

3. Every computed value is well under one second, so the RFC 6298 minimum of 1 second governs. On short paths the timer is almost always the floor, which is why fast retransmit matters: waiting for the RTO would cost thirty round trips.
4. Fast retransmit and fast recovery are in RFC 5681, September 2009. The trigger is three duplicate ACKs, chosen because one or two duplicates are commonly caused by simple reordering.
5. Selective acknowledgement is RFC 2018, October 1996, by Mathis, Mahdavi, Floyd and Romanow. It is negotiated in the SYN and carries three or four blocks per segment because of the 40-byte option limit.
6. The receive window field is 16 bits, capping the raw value at 65,535 bytes. Window scaling, RFC 7323, September 2014, obsoleting RFC 1323, adds a shift count of up to 14, giving a maximum window just under 1 GiB. The scale factor is exchanged only in the SYN and SYN-ACK; if either side omits it, no scaling is used for the whole connection.
7. Zero window handling: the sender must send a one-octet probe on the persist timer, backing off exponentially, so a lost window update cannot deadlock the connection.
8. Nagle's algorithm is RFC 896, "Congestion Control in IP/TCP Internetworks", John Nagle, 6 January 1984. Rule: if unacknowledged data is outstanding, buffer small writes until an ACK arrives or a full segment can be sent.
9. Delayed acknowledgement is in RFC 1122, October 1989. An ACK may be delayed by less than 500 ms, and there must be an ACK for at least every second full-sized segment.
10. The two together create a well-known stall. The sender holds a small final piece under Nagle. The receiver holds its ACK under delayed ACK. Neither moves until the delayed-ACK timer fires.
11. On Linux that timer is adaptive between about 40 ms and 200 ms, from `TCP_DELACK_MIN` and `TCP_DELACK_MAX`. A protocol that writes a header and a body in two separate calls can add 40 ms per exchange for no reason.
12. The fixes, in order of preference: write the whole message in one call or use `writev`; or set `TCP_NODELAY` to disable Nagle. Disabling Nagle is normal for request-response traffic and is the default in most RPC libraries.
13. Inspect with `ss -tin` on Linux, which prints rtt, rttvar, cwnd, ssthresh, retrans and the negotiated window scale for every socket.

28.6.6 WORDS - remember these

1. Sequence number — the count of bytes sent so far — a 32-bit field naming the first octet of this segment's payload.
2. Acknowledgement — a receipt for what arrived — a cumulative ACK naming the next octet expected, implying everything below it arrived.
3. SACK — a note saying which pieces beyond the gap arrived — the selective acknowledgement option of RFC 2018.
4. Fast retransmit — resending early because of repeated receipts — resending the missing segment after three duplicate ACKs, per RFC 5681.
5. RTO — how long to wait before assuming loss — the retransmission timeout of RFC 6298, SRTT plus four times RTTVAR, floored at 1 second.

28.7 Congestion control in plain words

28.7.1 PLAIN - in simple words

1. Flow control stops you drowning the person you are talking to. Congestion control stops you drowning the road between you. Mixing them up is one of the most common mistakes in this subject.
2. The receiver tells you its buffer size, so flow control is easy. Nobody tells you the capacity of the network, so the sender has to guess and keep re-guessing.
3. The guess is called the **congestion window**, a second limit on how much may be in flight, kept alongside the receiver's window.

4. The strategy is simple: start small, increase while things go well, cut hard when a packet is lost.
5. Loss is the signal, because on a plain IP network loss is the only message the network sends about being full.
6. That works beautifully when loss means congestion, and terribly when loss means a bad radio link, because then TCP slows for no reason.

28.7.2 PLAIN - a picture in your head

1. Imagine driving into an unfamiliar city at night with no speed signs.
2. You start slowly. Nothing bad happens, so you go faster. At some point you scrape a kerb. You brake hard, then creep back up.
3. That is congestion control: millions of senders, no central authority, all probing for a limit and backing off when they find it.
4. Now imagine your car reports a scrape whenever you drive over a bump. You crawl through an empty city, convinced it is full. That is TCP on a lossy wireless link.

Where this comparison breaks: a driver can see the road ahead. TCP cannot see anything. It learns about the road only by driving into it and noticing what happened one round trip later. Every decision is made with information that is already old.

28.7.3 PLAIN - a worked example

1. Slow start is badly named. It is not slow; it is a doubling. Start with a congestion window of 10 segments, the modern default.

Round trip	Window, segments	Bytes in flight
1	10	14,600
2	20	29,200
3	40	58,400
4	80	116,800
5	160	233,600
6	320	467,200

2. Doubling continues until a loss happens or the window passes the slow start threshold, after which it grows by about one segment per round trip. So the shape is rocket, then crawl, then a cut on loss, then crawl again.
3. Now the bandwidth-delay product: how much data must be in flight to keep a link busy, which is rate multiplied by round-trip time.

Path	RTT	Rate	Must be in flight
Laptop to Pune edge	30 ms	100 Mbit/s	375 KB
India to US East	250 ms	100 Mbit/s	3.1 MB
India to US East	250 ms	1 Gbit/s	31 MB

4. The 30 millisecond figure is the right order for the reader's own path, which ended at Microsoft routers in Pune. The 250 millisecond figure is typical for India to the east coast of the United States. Both are approximate.
5. Without window scaling the maximum window is 65,535 bytes. Divide that by the round-trip time to get the ceiling on speed: about 17 megabits per second at 30 milliseconds, and about 2.1 megabits per second at 250 milliseconds.
6. A gigabit connection from India to a US server, without window scaling, delivers about two megabits per second. The link is not the problem. The window is.

28.7.4 PLAIN - what is really happening inside

1. The sender keeps two numbers: `cwnd`, the congestion window, and `ssthresh`, the slow start threshold.
2. While `cwnd` is below `ssthresh`, the sender is in slow start and `cwnd` grows by one segment per segment acknowledged, which doubles it per round trip.
3. Above `ssthresh`, the sender is in congestion avoidance and `cwnd` grows by about one segment per round trip, however many acknowledgements arrive.
4. When three duplicate acknowledgements arrive, the sender halves `ssthresh`, sets `cwnd` near the new `ssthresh`, retransmits, and continues. That is fast recovery, a mild reaction.
5. When the retransmission timer fires instead, the reaction is severe: `cwnd` drops all the way back to one segment and slow start begins again. That is why fast retransmit was such an important invention.

28.7.5 TECHNICAL - the engineer's version

1. The problem was not theoretical. In October 1986 the link between Lawrence Berkeley Laboratory and the University of California, Berkeley, a distance of about 400 yards, collapsed from 32 kbit/s to 40 bit/s.
2. Van Jacobson diagnosed it and published "Congestion Avoidance and Control" at ACM SIGCOMM in August 1988, introducing slow start, congestion avoidance and the RTT variance estimator.
3. 4.3BSD-Tahoe, 1988, shipped slow start, congestion avoidance and fast retransmit. 4.3BSD-Reno, 1990, added fast recovery. The algorithm names come from the release names.
4. The current standard is RFC 5681, "TCP Congestion Control", September 2009, with NewReno loss recovery in RFC 6582, April 2012. The initial window of 10 segments comes from RFC 6928, April 2013; before that it was 2 to 4.
5. CUBIC is the Linux default since kernel 2.6.19 in November 2006, from the 2008 work of Sangtae Ha, Injong Rhee and Lisong Xu. It is now standards track as RFC 9438, August 2023, obsoleting the informational RFC 8312.
6. CUBIC grows the window as a cubic function of the time since the last reduction, making it far more aggressive than Reno on high bandwidth-delay-product paths and roughly as polite on short ones.
7. BBR, from Google in 2016 by Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh and Van Jacobson, takes a different view: it estimates the bottleneck bandwidth and the minimum round-trip time and paces to that, rather than treating loss as the signal.
8. Established fact: BBR usually outperforms loss-based algorithms on paths with non-congestive loss and deep buffers. Active dispute: how fairly BBR shares capacity with CUBIC in shallow buffers. BBR version 3 was presented in 2023 and is being standardized in the IETF; it is a draft, not an RFC.
9. The Mathis equation, from the 1997 paper by Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi and Teunis Ott, gives the throughput ceiling of a loss-based algorithm.

```
rate  <=  (MSS / RTT) * (C / sqrt(p))

MSS = 1460 bytes,  C = 1.22,  p = loss probability
```

Loss rate	RTT 30 ms	RTT 250 ms
0.01 percent	47 Mbit/s	5.7 Mbit/s
0.1 percent	15 Mbit/s	1.8 Mbit/s
1 percent	4.8 Mbit/s	0.57 Mbit/s
5 percent	2.1 Mbit/s	0.26 Mbit/s

10. Read the bottom-right cell. One in twenty packets lost on a long path gives you 260 kbit/s regardless of how fat the pipe is. This is why "the link is 1 Gbit/s" tells you almost nothing on its own.
11. Inspect and change with `sysctl net.ipv4.tcp_congestion_control` on Linux, `ss -tin` to see `cwnd` and `ssthresh` per socket, and `sysctl net.inet.tcp.cc.available` on macOS and FreeBSD.

28.7.6 WORDS - remember these

1. Congestion window — the sender's guess at what the road can take — `cwnd`, a sender-side limit measured in segments or bytes.
2. Bandwidth-delay product — how much must be in the air to keep a pipe full — rate multiplied by round-trip time, in bytes.
3. CUBIC — today's usual algorithm — the cubic-growth loss-based controller of RFC 9438, default on Linux since 2006.
4. BBR — the model-based alternative — a controller estimating bottleneck bandwidth and minimum RTT, from Google in 2016, still an IETF draft.

28.8 The three outcomes of a connection attempt

28.8.1 PLAIN - in simple words

1. This section is the heart of the chapter. Everything before it was preparation.
2. When you try to open a TCP connection, exactly three things can happen. Not four. Three.
3. **Success.** The far side answers with SYN-ACK. You are connected, in about one round trip.
4. **Refused.** Something answers, but the answer is no. You get an error almost immediately, well under a second on any normal path.
5. **Dropped.** Nothing answers. Not a yes, not a no, not an error. Silence. Your machine tries again a few times and eventually gives up.
6. Here is the rule to carry with you: **a refusal is a reply. A drop is not.** So the speed of a failure tells you which kind it was, before you know anything else about the network.
7. Fast failure means something on the path is alive, received your packet, and decided against you. Slow silent failure means your packet, or the answer to it, disappeared without anyone saying so.
8. The reader's failure was the slow silent kind. That one fact rules out a whole category of explanations, which is what section 28.9 does.

28.8.2 PLAIN - a picture in your head

1. You knock on a door.
2. Someone opens it and lets you in. That is success.
3. Someone shouts through the door "go away" and you leave immediately. That is refusal. Unpleasant, but fast, and it proves someone is home.
4. Nothing happens. You knock again. Nothing. Eventually you walk away. That is a drop.
5. Notice how much less you learned in case three. You do not know if the house is empty, if the bell is broken, if there is a wall you cannot see, or if someone inside is deliberately ignoring you.

Where this comparison breaks: with a real door you can look at the windows and check whether the lights are on. On a network you often cannot get a second opinion from the same vantage point, which is why testing from another network, as the reader did with mobile data, is worth more than any amount of retrying.

28.8.3 PLAIN - a worked example

```
You send:  SYN ------------------------------>  ?

(a) SUCCESS
    <-------------------------------- SYN, ACK
    result: connected in about one round trip

(b) REFUSED
    <-------------------------------- RST, ACK
      or  <----------------- ICMP dest unreachable
    result: error returned in well under a second
```

```
(c) DROPPED
    ...nothing...
    SYN ------------------------->  ...nothing...
    SYN ------------------------->  ...nothing...
    SYN ------------------------->  ...nothing...
    result: timeout after tens of seconds
```

Outcome	What comes back	Time	Typical message
Success	SYN-ACK	1 round trip	connection opens
Refused	RST, or ICMP	under 1 second	Connection refused
Dropped	nothing at all	15 to 130 s	Connection timed out

1. Try this on your own machine to feel the difference in speed.

```
# (b) refused: nothing is listening on port 9 locally
nc -vz 127.0.0.1 9
# nc: connect to 127.0.0.1 port 9 failed: Connection refused
# returns instantly

# (c) dropped: a documentation address that goes nowhere
nc -vz -w 30 192.0.2.1 443
# nc: connect to 192.0.2.1 port 443 failed: Operation timed out
# returns only after the full wait
```

2. The two commands differ only in the address. The difference in how long they take is the entire diagnostic.

28.8.4 PLAIN – what is really happening inside

1. In the refused case, a machine received your SYN, found no socket listening on that port, and replied with a TCP segment carrying the RST flag.
2. RST means reset. It says: this connection does not exist and never will.
3. Your kernel receives that RST while in SYN_SENT, moves the socket to CLOSED, and returns the error `ECONNREFUSED` to your program at once.
4. A firewall can also produce a refusal, by sending an ICMP destination unreachable message, most often code 3 for port unreachable or code 13 for administratively prohibited.
5. In the dropped case, no such packet exists. Your kernel sits in SYN_SENT with a retransmission timer running and nothing to react to.
6. The timer fires, the SYN is sent again, and the timer is doubled. Then again, and doubled again. This is exponential backoff.
7. After a fixed number of attempts, or a fixed total time, the kernel gives up and returns `ETIMEDOUT`.

28.8.5 TECHNICAL – the engineer's version

1. Errors as your program sees them: `ECONNREFUSED` for a refusal, `ETIMEDOUT` for a drop, `EHOSTUNREACH` or `ENETUNREACH` when ICMP or the local routing table says the destination cannot be reached at all.
2. RFC 9293 requires an RST in response to a SYN for a port with no listener, unless local policy says otherwise, with hardening rules in RFC 5961.
3. The relevant ICMP type is 3, Destination Unreachable, from RFC 792. Code 3 is port unreachable; codes 9, 10 and 13 are the administratively prohibited family added by RFC 1812, June 1995; code 4 is fragmentation needed.
4. Here is the Linux SYN retry schedule with the default `net.ipv4.tcp_syn_retries` of 6 and the RFC 6298 initial RTO of 1 second.

Attempt	Sent at	Then waits
SYN 1	0 s	1 s
SYN 2	1 s	2 s
SYN 3	3 s	4 s
SYN 4	7 s	8 s
SYN 5	15 s	16 s
SYN 6	31 s	32 s
SYN 7	63 s	64 s
give up	127 s	-

5. The Linux kernel documentation states this exactly: six retries corresponds to 63 seconds until the last retransmission, and a final timeout at 127 seconds.
6. The server side has its own counter, `net.ipv4.tcp_synack_retries`, default 5, giving 31 seconds to the last SYN-ACK retransmission and a final timeout at 63 seconds.
7. macOS and the BSDs use a total-time cap instead of a retry count. The sysctl is `net.inet.tcp.keepinit`, default 75000 milliseconds, so a connection attempt is abandoned after 75 seconds.

System	Control	Default	Gives up after
Linux	tcp_syn_retries	6	127 s
macOS, BSD	tcp.keepinit	75000 ms	75 s
Windows	retransmit count	2, approx	about 21 s

8. The Windows row is approximate, because the retransmission count and the 3 second initial timeout have changed across versions.
9. So a 5 second timeout in a log may be the application's limit, not the kernel's, and tells you less than a full kernel-length timeout does.
10. The mapping onto scanner vocabulary is exact: success is `open`, refusal is `closed`, and a drop is `filtered`. Section 28.12 returns to this.

28.8.6 WORDS - remember these

1. RST — the packet that means "no, and stop" — the TCP reset control bit, aborting a connection immediately.
2. Connection refused — someone answered no — the `ECONNREFUSED` error, caused by an RST or an ICMP port unreachable.
3. Connection timed out — nobody answered at all — the `ETIMEDOUT` error, caused by exhausting SYN retransmissions.
4. Exponential backoff — waiting twice as long after each failure — the doubling of the retransmission timer, 1, 2, 4, 8 seconds and onward.
5. Silent drop — a packet discarded with no notification — a filtering or forwarding decision that generates no ICMP and no RST.

28.9 What the reader's timeout actually proves

28.9.1 PLAIN - in simple words

1. Here is what the reader saw, exactly as recorded. `curl -v https://github.com` printed `Trying 20.207.73.82:443...`, then stopped for 15 seconds and failed with nothing coming back at all.
2. No refusal. No error message from any router. No reset. Silence.
3. Now let us extract every single thing that observation proves, and refuse to claim anything more.

4. **One. DNS worked.** The name `github.com` became the address `20.207.73.82`, so the resolver at `1.1.1.1` was reachable and answered.
5. **Two. Routing worked at least part of the way.** The traceroute reached hop 12, `ae106-0.rwa04.pnq20.ntwk.msn.net`, at `104.44.20.42`. Packets from this laptop were leaving the flat, crossing the ISP and entering Microsoft's network in Pune.
6. **Three. The TCP handshake never completed.** The connection never reached ESTABLISHED. It stayed in SYN_SENT until it was abandoned.
7. **Four. Nothing actively refused it.** No RST came back. If one had, curl would have said "Connection refused" in a fraction of a second.
8. **Five. No router complained either.** No ICMP destination unreachable arrived. If one had, curl would have failed quickly with a different message.
9. **Six. Therefore something swallowed a packet in silence.** Either the SYN never reached a machine willing to answer, or the SYN-ACK it produced never made it home. One of those two packets died without a word.
10. That is the complete list. Everything else people usually say about a timeout is guessing.

28.9.2 PLAIN – a picture in your head

1. You post a letter that needs a signature on delivery. Three weeks later you have no signature and no returned envelope.
2. Compare this with getting the envelope back stamped "no such address". That stamp would tell you a sorting office read your address and rejected it.
3. But silence is not useless. It definitively rules out every story that would have produced a stamp.

Where this comparison breaks: with post you could telephone the recipient and ask. The network equivalent is testing from a different vantage point, and the reader did exactly that with mobile data. That is why the mobile test was the most valuable single action in the whole session.

28.9.3 PLAIN – a worked example

1. Here is the evidence laid out as a ladder, from most certain to least.

Claim	Status	Basis
DNS resolved the name	proven	address was printed
Some path existed outward	proven	trace reached hop 12
Handshake did not complete	proven	curl timed out
Nothing sent an RST	proven	no refusal message
No ICMP error arrived	proven	no unreachable message
A packet was dropped silently	proven	follows from the above
The drop was after hop 12	not proven	see below
A named party blocked it	not proven	intent is invisible
The server was down	contradicted	mobile data worked

2. Note the line about hop 12. The traceroute used ICMP or UDP probes, not TCP packets to port 443, and networks routinely treat those differently. A device can forward ICMP happily and drop TCP SYNs to port 443, or the reverse.
3. So "the trace reached hop 12" proves that some packets from this laptop reached Pune. It does not prove that this SYN did.
4. The session record notes several `utun` interfaces on the macOS machine, which are tunnel interfaces created by VPN software.
5. A stale route pointing `20.207.73.82` into a `utun` interface whose tunnel is not up produces precisely this symptom: a SYN that goes nowhere and a silent timeout.

6. That is not a claim that this happened. It is a candidate the collected evidence cannot rule out, and one command would have settled it.

28.9.4 PLAIN - what is really happening inside

1. Walk the SYN's possible fates in order, from the laptop outward.
2. It could have been dropped by the local packet filter on macOS, or routed into a tunnel interface that leads nowhere because a VPN left a route behind.
3. It could have crossed the ISP's access network and been dropped in the private core at hops 2 to 6, which are carrier addresses in the RFC 1918 ranges.
4. Or, and this is the one people forget, it could have arrived perfectly and produced a SYN-ACK that was dropped on the way back. The return path is not necessarily the mirror of the outbound path.
5. From the client's chair, all of these look identical: silence for 15 seconds. That is the fundamental limitation, and no amount of retrying from the same chair fixes it.

28.9.5 TECHNICAL - the engineer's version

1. First, an honest note about the 15 seconds. The macOS kernel default is `net.inet.tcp.keepinit` at 75000 milliseconds, so the kernel would have kept trying for 75 seconds.
2. A stop at 15 seconds therefore came from something above the kernel: a `--connect-timeout` or `--max-time` flag, a `timeout` wrapper, or a tool default. It does not change the conclusion, but the number should be attributed correctly.
3. Second, the candidate causes, with the evidence that would separate them.

Candidate	Distinguishing test
Local filter or bad route	tcpdump on the wire interface
Stale VPN route via utun	route get 20.207.73.82
Port-based filtering	try 80 and 22 to same address
Address or prefix block	try another 20.x address
Routing black hole	TCP traceroute to port 443
Asymmetric return drop	test from a second network
MTU black hole	connects, then stalls later
Middlebox state exhaustion	intermittent, load dependent

4. Take the MTU case seriously, because it is the one that does not fit. A path MTU black hole occurs when a router must fragment, sends ICMP type 3 code 4 "fragmentation needed", and that ICMP is filtered. Path MTU Discovery is RFC 1191, November 1990; the probing alternative is RFC 4821, March 2007.
5. But a SYN is about 60 bytes. It fits through any MTU on earth. An MTU black hole therefore produces a connection that completes and then hangs, usually at the TLS ClientHello or the first large response.
6. The reader's symptom was the opposite: the connection never opened at all. So MTU is a poor fit here and should be ranked low. Naming it and then ruling it out is better practice than never mentioning it.
7. The single most informative missing command is a capture, which answers "did the SYN leave, and on which interface".

```
sudo tcpdump -n -i any \
  'host 20.207.73.82 and tcp port 443'

# expected on a drop, repeated with growing gaps:
# 12:00:00.000  192.168.0.101.51514 > 20.207.73.82.443: [S]
# 12:00:01.001  192.168.0.101.51514 > 20.207.73.82.443: [S]
# 12:00:03.005  192.168.0.101.51514 > 20.207.73.82.443: [S]
# and nothing at all coming back
```

8. The second most informative is a TCP-based traceroute, because it probes with the same protocol and port that is failing.

```
sudo traceroute -T -p 443 20.207.73.82
sudo nmap -sS -Pn -p 443 --traceroute 20.207.73.82
```

9. Now the rigour about what is not proven, stated flatly.
10. The evidence does not identify where the drop happened. A silent drop leaves no marker anywhere in the client's view.
11. The evidence does not prove intent. A drop can come from a deliberate policy, a misconfiguration, a failed line card, a full state table in a carrier-grade NAT, or a route withdrawn during a reconvergence.
12. The evidence does not name a responsible party, and this book will not name one. It is the reader's ISP and it is Microsoft's edge network, and neither fact establishes who dropped anything.
13. Nor does it establish that the condition was permanent. A single 15 second sample says nothing about five minutes later.
14. What the evidence does support, and supports strongly: a silent discard somewhere on that path in that moment, and not a server that was down.

28.9.6 WORDS - remember these

1. Black hole — a place packets go and never return — a forwarding state that discards traffic without generating ICMP.
2. Asymmetric routing — the way back is not the way there — a path where forward and reverse traffic traverse different links.
3. MTU black hole — big packets vanish, small ones do not — a path where fragmentation-needed ICMP is filtered, breaking Path MTU Discovery.
4. Vantage point — where you are standing when you measure — the network location from which an observation was taken, which bounds what it can prove.

28.10 Why "the server is down" was ruled out

28.10.1 PLAIN - in simple words

1. The first thing most people say when a site does not load is "the site must be down". Here that explanation is wrong, on two independent grounds.
2. **Ground one: the failure had the wrong shape.** A server that is down usually still answers something. It answers badly, but it answers.
3. If the web program has crashed but the machine is running, the machine's network software receives your SYN, finds no listener, and sends back a reset. You get "Connection refused" in milliseconds.
4. If the machine itself is off, the last router on that network usually cannot find it and sends back a "host unreachable" message. Again, fast.
5. If the site is behind a load balancer whose machines are all unwell, the load balancer answers you. It completes the connection and sends an error page.
6. None of those look like fifteen seconds of nothing.
7. **Ground two: it worked elsewhere at the same time.** The same site loaded instantly on the reader's phone over mobile data. Same destination, same moment, different path.
8. If the destination were down, the phone would have failed too. So the problem was not at the destination. It was on one particular path.

28.10.2 PLAIN - a picture in your head

1. You phone a shop and hear nothing at all. Not a ring, not an engaged tone, not a recorded message. Dead air for fifteen seconds.
2. A shop closed for the day still makes your phone ring. A shop out of business gives you "this number is not in service", promptly.

3. Dead air is not a property of the shop. Dead air is a property of the line.
4. Then your friend, standing beside you, phones the same number on a different network and speaks to an assistant. Now you know. The shop is fine.

Where this comparison breaks: phone networks are far more consistent about generating tones than IP networks are about generating errors. Much of why network faults are hard is that the internet's equivalent of the engaged tone is optional, and is frequently switched off.

28.10.3 PLAIN - a worked example

1. Here is the same user-visible failure under five different causes.

Cause	What comes back	How long
Web process crashed	TCP RST	milliseconds
Whole host powered off	ICMP host unreachable	under a second
Backends unhealthy	HTTP 502 or 503	fast, full reply
Overloaded, queueing	slow but complete reply	seconds, then data
Silent drop on path	nothing at all	until you give up

2. Only the last row matches what the reader saw.
3. A service the size of GitHub is fronted by many machines behind anycast addressing and load balancers. For every one of them to go silent at once, while a phone two feet away reached it instantly, is not a coherent story.

28.10.4 PLAIN - what is really happening inside

1. Consider a server whose web software has died. The kernel is still running, still owns the IP address, and still runs the TCP state machine.
2. Your SYN arrives, the kernel looks for a socket in LISTEN on port 443, finds none, and generates an RST. RFC 9293 requires that.
3. Now consider a load balancer with no healthy backends. It accepted your TCP connection before checking anything, completes TLS, reads your request, finds nowhere to send it, and returns 502 Bad Gateway or 503 Service Unavailable.
4. In every case some device took ownership of your packet and produced a response. Silence means no device took ownership, which is a different category of fault, living in the network rather than in the service.

28.10.5 TECHNICAL - the engineer's version

1. The general principle: **a service failure produces a response with the wrong content; a path failure produces no response.** Distinguishing them is the first branch of any network diagnosis.
2. Independent-vantage testing is the strongest cheap evidence available. The mobile-data test is a control experiment: it changes exactly one variable, the network path, and holds the destination constant.
3. The honest version: the mobile test is slightly weaker than it looks. Mobile data may resolve `github.com` to a different address entirely, since large services return different answers to different resolvers and regions.
4. To make it airtight, pin the same address from the other network.

```
curl -v --resolve github.com:443:20.207.73.82 https://github.com
nc -vz -w 10 20.207.73.82 443
```

5. There is one real exception, where a healthy service does go silent. A large provider under a volumetric attack may drop SYNs from an entire source prefix rather than answer them, which to the client looks like a broken path.

6. So the correct conclusion is not "the network operator is at fault". It is "the destination was serving, and packets on this path were being discarded without notification". Only the second claim is supported.

28.10.6 WORDS - remember these

1. Service failure — the thing you wanted is broken but reachable — an error response generated by a machine that received your request.
2. Path failure — you cannot reach the thing at all — loss or misdelivery in the network, typically with no error generated.
3. Control experiment — change one thing and hold the rest fixed — testing the same destination and port from a second network path.

28.11 Reading curl -v line by line

28.11.1 PLAIN - in simple words

1. `curl` fetches a URL. The `-v` flag makes it narrate what it is doing, naming each stage as it starts and finishes.
2. There are three kinds of line. A line beginning with * is curl talking about itself. Nothing on the network looks like that.
3. A line beginning with > is something curl sent to the server. A line beginning with < is something the server sent back, so the first < line is the first proof that the far side is really talking to you.
4. The reader's run stopped after `Trying 20.207.73.82:443...`, the line curl prints just before it starts the TCP handshake. So the name lookup finished and the handshake never did. The output pinpoints the stage.

28.11.2 PLAIN - a picture in your head

1. Think of curl's verbose output as a running commentary from a courier.
2. "Looking up the address." "Driving there." "Arrived at the gate." "Showing my papers." "Papers accepted." "Handing over the request." "Receiving a reply."
3. If the commentary stops after "driving there", the courier never arrived, and you stop worrying about the papers. If it stops after "papers accepted", the route and the gate were fine and the problem is later.

Where this comparison breaks: a courier reports its own progress honestly. curl only knows what the operating system tells it, and the operating system only knows what came back. Neither can report on a packet that vanished outbound. The commentary stopping is an absence of news, not news of a failure.

28.11.3 PLAIN - a worked example

1. A successful run, trimmed, with the meaning of each line beside it.

```
*   Trying 20.207.73.82:443...      starting TCP handshake
* Connected to github.com port 443  handshake done, ESTABLISHED
* ALPN: curl offers h2,http/1.1     offering HTTP versions in TLS
* TLSv1.3 (OUT), Client hello (1)   first TLS message sent
* TLSv1.3 (IN), Server hello (2)    first TLS message received
*  subject: CN=github.com           the name on the certificate
*  issuer: C=US; O=DigiCert Inc     who signed it
> GET / HTTP/2                      request line sent
> Host: github.com                  request header sent
< HTTP/2 200                        first byte back from server
< content-type: text/html           response header received
```

2. Now the reader's failing run, which is much shorter.

```
*   Trying 20.207.73.82:443...
* connect to 20.207.73.82 port 443 failed: Operation timed out
```

```
* Failed to connect to github.com port 443 after 15002 ms
* Closing connection 0
curl: (28) Failed to connect to github.com port 443
```

3. Two facts jump out. There is no `Connected to` line, so the handshake never finished. And there is no `<` line anywhere, so nothing came back at all.

Code	Name	Means
6	Could not resolve host	DNS failed
7	Failed to connect	usually refused
28	Operation timed out	silence
35	SSL connect error	TLS stage failed
60	Certificate problem	trust or name failed

4. Code 7 and code 28 are the refusal and the drop from section 28.8, wearing curl's clothing.

28.11.4 PLAIN – what is really happening inside

1. `curl -v` does not measure time. For numbers you use `-w`, the write-out option, which prints variables after the transfer.

```
curl -sS -o /dev/null -w '
  dns         %{time_namelookup}
  tcp         %{time_connect}
  tls         %{time_appconnect}
  ttfb        %{time_starttransfer}
  total       %{time_total}
  address     %{remote_ip}:%{remote_port}
' https://github.com
```

2. Every time value is measured from the start of the request, so they are cumulative, not durations. You subtract to get each stage.

Field	Example	Stage duration
time_namelookup	0.021 s	DNS took 21 ms
time_connect	0.055 s	TCP took 34 ms
time_appconnect	0.128 s	TLS took 73 ms
time_starttransfer	0.291 s	server took 163 ms
time_total	0.354 s	body took 63 ms

3. In the reader's case `time_namelookup` would be small and every later field would read zero. A resolved address with a `time_connect` of zero is the machine-readable signature of the drop case.

28.11.5 TECHNICAL – the engineer's version

1. `--connect-timeout` limits only the setup phase, which curl defines as DNS plus the TCP, TLS or QUIC handshakes. The libcurl default for `CURLOPT_CONNECTTIMEOUT` is 300 seconds.
2. `--max-time` limits the whole operation including the body. Use both: a short connect timeout and a longer overall one.
3. `-4` forces IPv4 and `-6` forces IPv6. This matters constantly on dual-stack networks, where a broken IPv6 path produces long stalls. The reader's machine reported `IPv6: (none)`, so `-4` would have changed nothing here.

4. `--resolve host:port:address` overrides DNS for one name and port, letting you test one specific server without changing any system configuration.
5. Here are the exact commands the reader should have run.

```
# 1. is it TCP, or is it TLS or HTTP above it
nc -vz -w 5 20.207.73.82 443

# 2. is it this port, or this address
nc -vz -w 5 20.207.73.82 80
nc -vz -w 5 140.82.121.4 443

# 3. is a name problem hiding as a network problem
dig +short github.com @1.1.1.1
curl -v --resolve github.com:443:20.207.73.82 https://github.com

# 4. did the SYN leave the laptop, and by which interface
sudo tcpdump -n -i en0 'host 20.207.73.82 and tcp port 443'
route get 20.207.73.82

# 5. how far does a TCP SYN to 443 actually get
sudo nmap -sS -Pn -p 443 --reason --traceroute 20.207.73.82

# 6. timing signature, in one line
curl -sS -o /dev/null -m 20 \
  -w 'dns %{time_namelookup} tcp %{time_connect}\n' \
  https://github.com
```

6. Command 1 separates the transport from everything above it: if `nc` also hangs, the fault is at or below TCP. Command 2 distinguishes a port-specific filter from an address-specific one, for ten seconds of effort.
7. Command 4 is the only one that can prove the packet left the machine, which is the assumption every other test quietly makes. Command 5 probes with a SYN to port 443, so unlike an ICMP traceroute it measures the failing path.

28.11.6 WORDS - remember these

1. Write-out — print measured values after the transfer — curl's `-w` option with variables such as `time_connect`.
2. Resolve override — pin a name to an address for one command — curl's `--resolve host:port:address`.
3. Exit code — the number the command returns — curl's documented failure codes, 7 for connect failure and 28 for timeout.

28.12 The other tools that answer this question

28.12.1 PLAIN - in simple words

1. `curl` tests a whole stack at once: DNS, TCP, TLS and HTTP. When it fails you still have to work out which layer failed.
2. `nc`, usually called netcat, opens a bare TCP connection and reports whether it opened. Nothing above TCP is involved.
3. `telnet` did the same job for decades, and old documentation still recommends it. It works, but modern systems are removing it.
4. `nmap` tries many ports at once and classifies each. Its vocabulary maps exactly onto the three outcomes of section 28.8.

5. `tcpdump` shows the actual packets on the wire, and is the only tool here that can prove a packet left your machine.

28.12.2 PLAIN - a picture in your head

1. Think of a doctor with four instruments.
2. `curl` is asking the patient how they feel: rich information, hard to localize. `nc` is a single specific test: does this one joint bend.
3. `nmap` is a broad screen: check twenty things quickly and flag the odd ones.
4. `tcpdump` is the X-ray: it shows what is actually there rather than what anyone reports. You would not start with it, and you would not finish without it when the reports disagree.

Where this comparison breaks: an X-ray sees inside the patient. `tcpdump` only sees packets passing your own machine's interfaces. It can prove your SYN left; it can never show you what a router 2,000 kilometres away did with it.

28.12.3 PLAIN - a worked example

```
# netcat, bare TCP connect, 5 second limit
nc -vz -w 5 20.207.73.82 443
# open:     Connection to 20.207.73.82 port 443 succeeded
# closed:   nc: connect to ... failed: Connection refused
# filtered: nc: connect to ... failed: Operation timed out
```

```
# nmap, with the reason printed
sudo nmap -sS -Pn -p 443 --reason 20.207.73.82
# PORT    STATE     SERVICE  REASON
# 443/tcp open      https    syn-ack ttl 52
# 443/tcp closed    https    reset ttl 52
# 443/tcp filtered  https    no-response
```

1. `syn-ack` means a handshake reply arrived. `reset` means an RST arrived. `no-response` means nothing arrived. That is section 28.8, word for word.

Outcome	nmap state	nmap reason
Success	open	syn-ack
Refused	closed	reset or ICMP
Dropped	filtered	no-response

28.12.4 PLAIN - what is really happening inside

1. `nc -z` and `telnet` perform a normal `connect` call through the operating system, so they see exactly what any ordinary program would see.
2. `nmap -sS` builds raw packets itself, sends a SYN, reads the reply, and never completes the handshake. That is why it needs administrator rights and is called a half-open or stealth scan. `nmap -sT` uses `connect` instead.
3. `tcpdump` asks the kernel for a copy of packets at the interface, so it shows outbound packets even when nothing comes back. A capture showing three or four outgoing SYNs with growing gaps and nothing inbound is direct proof of the drop case.

28.12.5 TECHNICAL - the engineer's version

1. Telnet is RFC 854, May 1983, by Jon Postel and Joyce Reynolds. Apple removed the `telnet` and `ftp` clients from macOS in version 10.13 High Sierra, September 2017. On Windows the Telnet Client has been an optional feature, off by default, since Windows Vista. It is unencrypted and must never be used for real remote login; as a reachability probe `nc` is better in every way.

2. Nmap's authoritative state definitions: `open` means an application is actively accepting connections; `closed` means the port is reachable but no application is listening; `filtered` means packet filtering prevents probes from reaching the port, so nmap cannot determine the state.
3. Always pass `-Pn`. Without it, a host that ignores pings is recorded as down and never scanned at all, which yields a misleading empty result.

```
# only SYN packets, either direction, to one host
sudo tcpdump -n 'host 20.207.73.82 and tcp[tcpflags] & tcp-syn != 0'

# only resets, useful for spotting refusals
sudo tcpdump -n 'tcp[tcpflags] & tcp-rst != 0'

# all ICMP, for unreachables and MTU messages
sudo tcpdump -n icmp

# write a file for later analysis in Wireshark
sudo tcpdump -n -w capture.pcap 'host 20.207.73.82'
```

4. Complementary socket-state tools: `ss -tan state syn-sent` on Linux shows every connection stuck in exactly the reader's state; on macOS use `netstat -an -p tcp` and look for `SYN_SENT`. `lsof -i -P -n` maps sockets to process names on both.

28.12.6 WORDS - remember these

1. netcat — a bare TCP connect tester — `nc`, opening a socket and reporting the result without any higher protocol.
2. Filtered — nmap's word for silence — a port state where probes get no response, matching the dropped outcome.
3. Packet capture — recording the actual packets — `tcpdump` or Wireshark reading frames from an interface, selected with a filter such as `tcp port 443`.

28.13 Sockets from the programmer's side

28.13.1 PLAIN - in simple words

1. Everything in this chapter is reached from a program through about eight function calls, with the same names in almost every language.
2. `socket` creates an endpoint and returns a small number, a handle. `bind` attaches that endpoint to a local address and port. Servers do this; clients usually skip it and let the system choose.
3. `listen` turns a server endpoint into a door and sets how many callers may queue. `accept` takes one waiting connection off that queue and gives you a new handle for that one client.
4. `connect` is the client's call. It sends the SYN and waits for the handshake. This is the call that took 15 seconds and failed for the reader.
5. `send` and `recv` move bytes. On a stream socket they may move fewer bytes than you asked for, and you must handle that.
6. `close` releases the handle and starts the four-way close. And separately you set a timeout, because without one your program inherits the operating system's very long default.

28.13.2 PLAIN - a picture in your head

1. `socket` is renting a telephone. `bind` is being assigned an extension number.
2. `listen` is telling the switchboard to put callers through, with a queue of at most so many people on hold. `accept` is picking up the next caller from that queue; the main line stays free for the next one.
3. `connect` is dialling out. `send` and `recv` are talking and listening, and `close` is the polite mutual goodbye of section 28.5.

Where this comparison breaks: a phone conversation preserves the boundaries between sentences. A TCP stream does not. If you `send` three times, the other side may `recv` all of it in one go, or in seven pieces. Every real protocol therefore carries its own length prefix or delimiter.

28.13.3 PLAIN - a worked example

1. A minimal server, in C, that echoes back whatever it is sent.

```
int s = socket(AF_INET, SOCK_STREAM, 0);

struct sockaddr_in a;
memset(&a, 0, sizeof a);
a.sin_family      = AF_INET;
a.sin_addr.s_addr = htonl(INADDR_ANY);
a.sin_port        = htons(8080);

int one = 1;
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &one, sizeof one);

bind(s, (struct sockaddr *)&a, sizeof a);
listen(s, 128);            /* 128 = backlog queue length */

for (;;) {
    int c = accept(s, NULL, NULL);   /* one client */
    char buf[1024];
    ssize_t n = recv(c, buf, sizeof buf, 0);
    if (n > 0) send(c, buf, (size_t)n, 0);
    close(c);              /* this side does the active close */
}
```

2. A client with a timeout, in Python, that reports which of the three outcomes happened.

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(5.0)          # applies to connect and to I/O

try:
    s.connect(("20.207.73.82", 443))
    print("SUCCESS: handshake completed")
except ConnectionRefusedError:
    print("REFUSED: an RST came back, something said no")
except socket.timeout:
    print("DROPPED: nothing came back at all")
except OSError as e:
    print("OTHER:", e)
finally:
    s.close()
```

3. Those three branches are exactly the three outcomes of section 28.8, in twelve lines of code.

28.13.4 PLAIN - what is really happening inside

1. `socket` allocates a kernel structure and a file descriptor, and `bind` writes a local address and port into it. Neither sends any packet.
2. `listen` moves the socket into LISTEN state and creates two queues: one for half-open connections awaiting the final ACK, one for completed connections awaiting `accept`, which hands one over as a new descriptor.

3. The number passed to `listen` is the backlog. If the completed queue is full, new connections are dropped or reset, and clients see a timeout or a refusal.
4. `connect` builds and sends the SYN, puts the socket in SYN_SENT, and sleeps until the SYN-ACK arrives, an RST arrives, or the timer runs out.
5. `send` copies your bytes into the kernel send buffer and returns without waiting, so a successful `send` proves nothing about delivery. `recv` copies out of the receive buffer and returns 0 exactly when the peer has sent a FIN.
6. `close` releases the descriptor and begins the FIN exchange. `shutdown` is the finer tool: it closes one direction while leaving the other open.

28.13.5 TECHNICAL - the engineer's version

1. The Berkeley sockets interface first shipped in 4.2BSD in August 1983 and is standardized in POSIX.1-2001. That is why network code looks the same in C, Python, Go, Java and Rust.
2. Timeouts are the part people get wrong. `SO_RCVTIMEO` and `SO_SNDTIMEO` apply to data operations. On Linux they do not reliably bound `connect`.
3. The portable way to bound `connect` is to set the socket non-blocking, call `connect`, expect `EINPROGRESS`, wait on `select` or `poll` for writability with your own deadline, then read `SO_ERROR` for the result.
4. Linux additionally offers `TCP_USER_TIMEOUT` for established connections and `TCP_SYNCNT` to override `tcp_syn_retries` per socket, which is the clean way to make one connection give up in 5 seconds instead of 127.

Option	Level	Effect
SO_REUSEADDR	SOL_SOCKET	bind despite TIME_WAIT
SO_KEEPALIVE	SOL_SOCKET	probe idle connections
TCP_NODELAY	IPPROTO_TCP	disable Nagle
TCP_SYNCNT	IPPROTO_TCP	SYN retries, Linux only

5. A successful return from `send` means the data is in your kernel's buffer. It does not mean it left the machine, arrived, or was read by the far end.
6. This is the same principle the reader met in git: a disconnect while reading a response tells you nothing about whether the write landed. The only way to know is to ask again.

28.13.6 WORDS - remember these

1. EINPROGRESS — "started, not finished" — the error returned by `connect` on a non-blocking socket, the basis of a custom connect timeout.
2. shutdown — close one direction only — the call that sends a FIN without releasing the descriptor.
3. TCP_NODELAY — send small writes at once — the socket option that disables Nagle's algorithm, standard in request-response systems.

28.98 Common wrong ideas

1. Wrong: a timeout means the server is down. Right: a timeout means nothing came back. A down server usually sends a reset or produces a fast error page. Silence points at the path, not at the service.
2. Wrong: if `ping` works, the site should work. Right: `ping` uses ICMP, not TCP. A device can forward ICMP and drop TCP port 443, and many do. `ping` failing also proves nothing, because ICMP is often blocked on purpose.
3. Wrong: TCP guarantees delivery. Right: TCP guarantees that data arrives complete and in order, or that the connection fails and you are told. It turns silent loss into correct delivery or a reported error. It cannot deliver across a cut cable.

4. Wrong: `closed` and `filtered` are the same thing. Right: `closed` means something replied with a reset, which proves the host is reachable. `filtered` means nothing replied, which proves almost nothing. They differ in evidence and in how long they take.
5. Wrong: UDP is just a worse TCP. Right: UDP is TCP with the reliability removed on purpose, so applications that cannot use late data are not forced to wait for it. Live voice, video calls, DNS and NTP all want it.
6. Wrong: thousands of TIME_WAIT sockets are a leak. Right: TIME_WAIT is normal for whoever closes first, and lasts 60 seconds on Linux. Thousands of CLOSE_WAIT sockets, by contrast, usually are a real application bug.
7. Wrong: `* * *` at the end of a traceroute proves the fault is at that hop. Right: many routers deprioritise or suppress ICMP time-exceeded replies. Silence at the end of a trace is the normal case, not a diagnosis.
8. Wrong: a successful `send` means the other side received the data. Right: it means the bytes are in your own kernel's buffer. Delivery is confirmed only by an acknowledgement you cannot see, or by a reply from the application.
9. Wrong: raising the timeout will fix a connection that times out. Right: a longer timeout only makes you wait longer for the same silence. It helps only when the path is slow, not when it is discarding packets.

28.99 Chapter summary in 20 lines

1. IP promises only to try. It gives no delivery, no ordering, no duplicate suppression, and no notification when a packet is lost.
2. TCP turns that into a reliable, ordered byte stream, and UDP is the layer that deliberately does not.
3. UDP is not a worse TCP. It is right whenever late data is worthless, such as live voice, live video, DNS queries and clock synchronization.
4. A port is a 16-bit number, 0 to 65,535, naming which program a packet is for.
5. A connection is identified by a four-tuple: source IP, source port, destination IP, destination port. That is why one server port serves millions.
6. A listening socket is matched on two values; a connected socket is matched on all four. Confusing them is the root of many bugs.
7. Port 443 is HTTPS, and it became the universal port because outbound 443 is nearly always permitted and its contents cannot easily be inspected.
8. Opening a connection costs a three-way handshake, SYN, SYN-ACK, ACK, and one full round trip before any application byte moves.
9. Initial sequence numbers are chosen unpredictably, per RFC 6528, because predictable ones allow blind injection attacks.
10. TCP is defined today by RFC 9293, STD 7, August 2022, which replaced the original RFC 793 of September 1981.
11. A connection walks eleven states. Closing takes four packets because each direction closes separately, and the active closer waits in TIME_WAIT.
12. Reliability comes from numbering every byte, cumulative acknowledgements, selective acknowledgement, retransmission timers, and fast retransmit on three duplicate acknowledgements.
13. Speed comes from the sliding window; the receiver's window is flow control and the sender's congestion window is congestion control. They are different.
14. Without window scaling, 65,535 bytes over a 250 millisecond path caps you at about 2 megabits per second regardless of link speed.
15. Loss-based congestion control treats every loss as congestion, so one percent loss on a long path cuts throughput to a fraction of a megabit per second.
16. A connection attempt has exactly three endings: SYN-ACK and success, RST or ICMP and a fast refusal, or silence and a slow timeout.
17. The diagnostic rule: fast failure means something answered; slow silent failure means something swallowed it.

18. The reader's `curl -v https://github.com` printed `Trying 20.207.73.82:443` and returned nothing for 15 seconds, proving DNS worked, some path existed as far as hop 12, the handshake never completed, and nothing refused it.
19. It does not prove where the drop happened, who caused it, or that it was deliberate. It does contradict "the server is down", because the same site loaded instantly on mobile data at the same moment.
20. `nc` isolates the transport, `nmap` names the outcome as open, closed or filtered, and `tcpdump` is the only tool that can prove the SYN ever left the machine.

29

CHAPTER

KB

Why Mobile Data Worked, and the Diagnostic Method

29.0 What this chapter gives you

1. You will be able to state the difference between "the site is down" and "I cannot reach the site", and say who is affected by each.
2. You will be able to give a one-line test that separates those two claims, and explain why one success anywhere beats a hundred failures here.
3. You will be able to list every single thing that changed when the reader's phone switched from home broadband to mobile data, and there are more than a dozen.
4. You will be able to explain why that test proved the destination was alive and yet proved nothing at all about which difference mattered.
5. You will be able to run a controlled experiment on a network: one variable at a time, with a stated hypothesis and a stated eliminating power.
6. You will be able to climb a seven-rung diagnostic ladder from "is my cable plugged in" to "where exactly does the path stop", and say for each rung what a pass proves, what a pass does not prove, and what a failure means.
7. You will be able to map the reader's own real fault onto that ladder and name the small set of causes that survive the evidence.
8. You will be able to use `ping`, `traceroute`, `dig`, `nc`, `curl`, `mtr` and a dozen other macOS commands with the flags that matter, and know what each one can and cannot prove.
9. You will be able to write up a fault so that an internet provider acts on it instead of asking you to reboot your router.
10. You will be able to catch an intermittent fault with a logging script, and tell a regional outage from a personal one.

29.1 "The site is down" and "I cannot reach the site"

29.1.1 PLAIN - in simple words

1. These two sentences sound like the same complaint. They are not.
2. "The site is down" is a claim about the whole world. It says nobody can use it. Not you, not your friend, not a person in another country.
3. "I cannot reach the site" is a claim about you. It says that from where you are sitting, right now, with this machine and this connection, it fails.
4. The second claim is small and almost always true when you say it. You did try. It did fail. There is nothing to argue about.
5. The first claim is enormous. To be true, it has to hold for every person on the planet at the same moment.
6. You almost never have the evidence for the first claim. You only ever have evidence for the second.

7. So when you say "GitHub is down" after one failed page load, you have quietly upgraded a small true statement into a huge unproven one.
8. That upgrade is what wastes the hours. Once you believe the site is down, you stop testing. Waiting feels like the correct action.
9. And if the real fault is on your own path, waiting fixes nothing, because nobody at the other end knows anything is wrong.
10. The whole method in this chapter is built to stop you making that jump.

29.1.2 PLAIN - a picture in your head

1. Imagine you post a letter to a friend in another city, and no reply comes.
2. You could say "my friend has moved away". That is a claim about your friend.
3. Or you could say "my letter did not get a reply". That is a claim about one letter, on one day, through one postal route.
4. The second is what you actually know. The first is a guess dressed up as knowledge.
5. There are many ways to get silence. The friend moved. The friend is ill. The letter was lost. The postbox at the end of your street is broken. A sorting office is on strike. The address on the envelope was fine but a customs check quietly binned it.
6. Now here is the trick that settles it in one move. You ask a person in another city to post the same letter to the same address.
7. If they get a reply, your friend has not moved. That is now certain. The problem is somewhere on your route, not at the far end.
8. You still do not know which part of your route. But you have cut the space of possible causes roughly in half, with one action.

Where this comparison breaks: letters are slow and rare, so you only get to test once or twice. Network tests are cheap and instant, so you can run twenty of them in a minute and narrow the cause much further than a letter ever could. Also, a letter has one route. A network packet may take a different route on every attempt, which is why an intermittent fault is so much harder than a missing friend.

29.1.3 PLAIN - a worked example

1. Here is the reader's real situation, in the two forms.
2. Weak form, the one that wastes time: "GitHub is down."
3. Strong form, the one that leads somewhere: "From my Mac, on my home broadband in India, `curl` to `github.com` on port 443 gets no response of any kind and gives up after 15 seconds."
4. The strong form names the machine, the network, the destination, the port, the tool, the symptom and the time. Every one of those is a variable you can change to learn something.
5. Now the one-line test that separates the two claims.

```
# Run this from a network that shares nothing with yours.
# Mobile data on a phone, or a friend's connection.
curl -sS -o /dev/null -w '%{http_code}\n' \
     --max-time 10 https://github.com
```

6. If that prints `200`, the site is not down. That is settled, permanently, by a single observation. No amount of failing from your desk can undo it.
7. If it also hangs and prints nothing, then the site being down has become a live possibility, and now a status page is worth reading.
8. The reader did exactly this test, in the simplest possible way. The same phone, switched from home Wi-Fi to mobile data, loaded `github.com` instantly.
9. So the claim "GitHub is down" was dead within about ten seconds. Everything after that was about finding the reader's own path problem.

29.1.4 PLAIN - what is really happening inside

1. There is a piece of plain logic underneath this, and it is worth learning once, because it applies far beyond networks.
2. "The site is down" means "for all people, the site fails". That is called a universal claim. It talks about everybody.
3. "I cannot reach the site" means "there exists at least one person, me, for whom it fails". That is called an existential claim. It talks about one case.
4. To prove a universal claim you would have to check everybody. You cannot.
5. To destroy a universal claim you need one counterexample. One person who can reach it. That is cheap, and you can do it in ten seconds.
6. To prove an existential claim you need one example. Your own failure is that example. So your own claim is already proved.
7. This gives an unfair-looking rule that is completely correct: a single success anywhere beats any number of failures in one place.
8. It also gives the reverse warning: no number of failures from your desk ever proves the site is down. Not ten, not a thousand. They are all the same observation repeated.
9. So the cheapest useful action in any connectivity fault is to buy one counterexample from a different vantage point.
10. A vantage point is just a place to test from: another network, another machine, another country, another provider.

29.1.5 TECHNICAL - the engineer's version

1. The distinction maps onto a standard split in incident work: a service-side fault versus a client-side or path-side fault.
2. A service-side fault has a blast radius covering all or a defined subset of users, is visible to the operator's own monitoring, and appears on a status page. Only the operator can fix it.
3. A path-side fault has a blast radius bounded by a network prefix, an autonomous system, a region or a single subscriber line. The operator's monitoring usually shows nothing at all.
4. This asymmetry is why large operators publish per-region status, and why "all systems operational" on a status page is compatible with your total inability to reach the service.
5. The logic in block 4 is Karl Popper's asymmetry between verification and falsification, set out in *Logik der Forschung* in 1934 and in English as *The Logic of Scientific Discovery* in 1959. Universal statements cannot be verified by observation but can be falsified by one counterexample.
6. The operational form of that rule is the single-vantage-point problem. One observer cannot distinguish global outage from local unreachability.
7. Distributed measurement platforms exist precisely to fix this. RIPE Atlas, launched by the RIPE Network Coordination Centre in 2010, runs a global fleet of small hardware and software probes, on the order of ten thousand active at any time, that anyone can task with ping, traceroute, DNS and TLS measurements from many countries at once.
8. Commercial equivalents include ThousandEyes, Catchpoint and Pingdom. Free web-based checkers do a weaker version of the same thing.
9. Terms to use precisely when reporting:

Phrase	What it actually claims
Service is down	Fails for all users
Partial outage	Fails for a named subset
Unreachable from AS X	Fails from one network
Degraded	Works, but slow or lossy
Cannot reproduce	Works from the operator

10. The verb matters. "Down" is a statement about the server. "Unreachable" is a statement about a pair: a source and a destination. Reachability is always a property of a pair, never of a single host.
11. That single sentence, that reachability is a property of a pair, is the most useful idea in this chapter.

29.1.6 WORDS - remember these

1. Down — nobody can use it — a service-side failure affecting all or a defined subset of users, visible in the operator's own monitoring.
2. Unreachable — I cannot get to it from here — a property of a source and destination pair, not of the destination alone.
3. Vantage point — a place you test from — an independent measurement source with its own address, provider and path.
4. Universal claim — a statement about everybody — falsifiable by a single counterexample, not verifiable by observation.
5. Existential claim — a statement about at least one case — proved by a single example, which your own failed attempt already supplies.
6. Blast radius — how many people a fault hits — the set of users, prefixes or regions affected by a given failure.

29.2 Everything that changed when the phone used mobile data

29.2.1 PLAIN - in simple words

1. The reader's test was simple. Same phone. Turn off Wi-Fi. Turn on mobile data. Load `github.com`. It loaded instantly.
2. It is tempting to read that as "so the problem is my Wi-Fi" or "so the problem is my router". Both may be wrong.
3. Switching to mobile data does not change one thing. It changes almost everything between you and the destination.
4. Here is the honest list of what changed, and there are more items than most people expect.
5. The company carrying your traffic changed. Home broadband company out, mobile company in.
6. Your address on the internet changed. Every reply now comes back to a different number.
7. The network that owns that address changed, which means the route the reply takes back to you is completely different too.
8. The physical path changed. Different cables, different cities, different handover point into the destination's network.
9. The name lookup probably changed. The phone on mobile data usually asks the mobile company's name server, not the one your Mac was using.
10. Because the lookup changed, the address you got back may have changed, so you may have talked to a different machine at the far end entirely.
11. The boxes in the middle changed. Home router, home firewall and your provider's inspection equipment are all out of the picture. The mobile company's own equipment is in it.
12. The kind of address may have changed. Mobile networks in India commonly give you a modern address type that home broadband often does not.
13. And a small one that matters more than people think: time passed. The second test happened minutes after the first.
14. Only one of those needs to be the cause. The test cannot tell you which.

29.2.2 PLAIN - a picture in your head

1. Imagine your parcel never arrives, so you try a different courier.
2. The second courier delivers it in an hour. You now know the address is real and somebody is there to sign for it.

3. That is genuinely valuable. It ends the argument about whether the recipient exists.
4. But look at what else changed when you switched courier. A different van, a different driver, a different depot, a different route, a different customs agent, a different set of rules about what they will carry, and a different day.
5. If someone now says "so the first courier's van was broken", they have invented an explanation. The van is one of seven things that changed.
6. Maybe the first courier refuses parcels over a certain size. Maybe their depot in one city is on strike. Maybe their scanner rejects your postcode.
7. You cannot tell. You changed everything at once, so the experiment has no resolving power about the cause.
8. It has enormous power about one specific question, and that question is "does the recipient exist". That question is now closed.

Where this comparison breaks: couriers deliver or fail visibly, and you get a tracking page. A network path can fail silently, with no message to anybody, and neither end learns that a packet was dropped. The far end in the reader's case never knew a connection had been attempted at all.

29.2.3 PLAIN - a worked example

1. Let us do the enumeration properly for the reader's case, item by item.
2. On home broadband, the Mac had a private address behind the router at `192.168.0.1`, and used the public name server `1.1.1.1`.
3. `github.com` resolved to `20.207.73.82`, which is an address owned by Microsoft, the company that has owned GitHub since 2018.
4. The connection attempt to that address on port 443 got no reply of any kind.
5. On mobile data, at minimum, the following changed.

What changed	Home broadband	Mobile data
Carrier	Fixed-line provider	Mobile operator
Public address	One number	A different number
Owning network	ISP's own network	Operator's network
Name server	1.1.1.1	Usually operator's
Answer for github.com	20.207.73.82	Possibly different
Address family	IPv4 only, no IPv6	Often IPv6 first
Middle boxes	Home router, ISP kit	Operator kit only
Physical path	Fixed line to city	Radio to core
Device stack	macOS	Phone OS
Client software	curl	Browser or app
Time of test	First attempt	Minutes later

6. That is eleven differences, and a careful person could find more.
7. Now the key sentence. Only one of those eleven needs to be the cause.
8. The test therefore proves exactly one thing: at that moment, some client somewhere could complete a request to `github.com`, so the service was up.
9. It does not prove your Wi-Fi is bad. It does not prove your router is bad. It does not prove your provider is blocking anything. It does not prove the address was different.
10. Those are all still open. The test closed one question and left eleven.

29.2.4 PLAIN - what is really happening inside

1. Let us go through why each change could, on its own, produce the observed result.
2. Different provider. Providers buy connectivity from different upstream networks. If one of the reader's provider's upstream links to the destination network is broken, packets die there and nowhere else.
3. Different public address. Some destinations block or rate-limit ranges of addresses. If the reader's home range was on such a list, packets would be dropped at the far end with no reply. That behaviour is normal for abuse controls.
4. Different owning network. The reply has to find its way back to you. If the return route from the destination network to your provider is broken, your request may arrive perfectly and the answer never comes home. You cannot tell the difference from your side.
5. Different name server. A different name server may hand back a different address for the same name, because large services steer users to a nearby machine based on where the question came from.
6. Different address, therefore a different machine at the far end. If the original machine was in trouble and the new one was not, the fault is at the destination but only for one of its many front doors.
7. Different address type. If mobile data used the newer, longer kind of address and home broadband had none, then the successful request travelled over a completely separate network layer, sharing almost nothing with the failed one.
8. Different middle boxes. Equipment that inspects traffic can drop a connection silently based on the name being requested, the destination address, the port, or a pattern inside the first packet.
9. Different time. A brief routing problem lasting a few minutes looks exactly like a permanent block if you only test twice.
10. So the correct conclusion from the mobile-data test is one sentence: the destination service is alive and serving at least some clients, and the fault lies in something that differs between the two attempts.
11. Anything more specific than that is a guess. Guesses are allowed, but they must be labelled as guesses and then tested one at a time.

29.2.5 TECHNICAL - the engineer's version

1. Enumerated precisely, the variables changed by a Wi-Fi to cellular switch are as follows.
2. Access technology: IEEE 802.11 wireless local area network to 4G LTE or 5G New Radio, which means a different link layer, a different maximum frame size and a different scheduler.
3. Source address: a new public IPv4 address, almost always behind carrier grade network address translation on the mobile side.
4. Autonomous system: the autonomous system number, the identifier for a network under one routing policy, changes. The reader's fixed-line provider and mobile operator hold different numbers.
5. Border Gateway Protocol path: both the forward path and, critically, the return path change. The return path is chosen by the destination network based on the new source prefix and is invisible from the client.
6. Peering relationship: the point where traffic hands off into Microsoft's network changes. The reader's fixed-line trace entered Microsoft at `ae66-0.del01-96cbe-1b.ntwk.msn.net`, a Delhi edge router. A mobile operator may hand off in Mumbai, Chennai or through a different transit provider altogether.
7. Recursive resolver: from `1.1.1.1`, Cloudflare's public resolver launched on 1 April 2018, to the mobile operator's own resolver, reached through the cellular data profile.
8. Resolved address: GitHub fronts through Microsoft edge, and the answer depends on the querying resolver's location and on any EDNS Client Subnet information, defined in RFC 7871. A different resolver can and often does yield a different anycast front end.
9. Address family: the session reported `IPv6: (none)` on the fixed line. Indian mobile networks commonly run IPv6-only bearers with 464XLAT, defined in RFC 6877 from April 2013, so the phone may have used a AAAA record and an entirely separate IPv6 path.
10. Middlebox set: home NAT, provider transparent proxies, deep packet inspection appliances and any Server Name Indication filtering differ completely between a fixed-line and a mobile bearer.

11. Path maximum transmission unit: typical fixed-line paths carry 1500 or 1492 bytes; cellular bearers frequently present 1400 to 1428 bytes, which changes fragmentation and TCP maximum segment size behaviour.
12. TCP and TLS fingerprint: a different operating system produces a different initial window, a different option order and a different TLS ClientHello. Filtering devices that key on ClientHello fingerprints will treat them differently.
13. Temporal variable: the two observations are not simultaneous, so any transient routing event separates them.
14. In experimental terms this is a confounded comparison. Thirteen variables moved together, so the experiment has a very high power to answer one binary question and near-zero power to localize a cause.
15. The formally correct conclusion is exactly this: the destination service was reachable and functional from at least one vantage point at time T, therefore the hypothesis "service down" is falsified for time T.

29.2.6 WORDS - remember these

1. Autonomous system — one company's network under one policy — a routing domain identified by an autonomous system number and announced in BGP.
2. Return path — how the answer comes back — the route chosen by the remote network towards your source prefix, invisible from the client side.
3. Peering point — where two networks hand traffic over — a physical or virtual interconnection governed by a peering or transit agreement.
4. Anycast front end — many machines sharing one address — the same address announced from multiple locations, with routing picking the nearest.
5. Confounded comparison — too many things changed at once — an experiment in which multiple independent variables move together, so no single cause can be attributed.
6. Middlebox — a box in the path that does more than forward — any device that inspects or alters traffic, such as a NAT, proxy, firewall or DPI appliance.
7. 464XLAT — a way to run IPv4 apps on an IPv6-only phone network — RFC 6877, combining a customer-side translator with a provider-side NAT64.

29.3 The logic of a controlled experiment on a network

29.3.1 PLAIN - in simple words

1. A controlled experiment is an old and simple idea. Change one thing. Keep everything else the same. See what happens.
2. If the result changes, the thing you changed is involved. If it does not, the thing you changed is probably innocent.
3. That is it. There is no more to the method than that sentence.
4. Networking breaks the rule constantly, because the easy tests change many things at once. Switching to mobile data changes thirteen things.
5. So the discipline is to design tests that move one thing and hold the rest still.
6. Before you run a test, say out loud what each possible result would mean.
7. If you cannot say what a pass would prove and what a fail would prove, the test is not worth running yet. Think first, type second.
8. This book gives that idea a name: the **eliminating power** of a test. It means how many possible causes a test can strike off your list.
9. A good test has high eliminating power. It strikes off half the list whichever way it goes.
10. A weak test is one where a pass tells you nothing and only a fail helps, or the other way round.

11. And there is one rule you must never break. Never conclude more than the evidence supports. Write down what you proved, not what you suspect.

29.3.2 PLAIN - a picture in your head

1. Think of the game where someone thinks of a number between 1 and 100 and you have to find it, and they only answer yes or no.
2. A bad question is "is it 37". Usually the answer is no, and you have removed one number out of a hundred.
3. A good question is "is it more than 50". Whatever the answer, half the numbers vanish.
4. Seven good questions find any number in 1 to 100. Ninety-nine bad questions might still miss.
5. Fault finding is that game. The list of possible causes is the numbers. Each command you type is a question.
6. Your job is to ask the question that halves the list, not the question that confirms your favourite theory.
7. And notice something about the game. You are only allowed to trust the answers you actually got. If they said "more than 50", you may not write down "so it is 73".

Where this comparison breaks: in the number game each answer is truthful and final. Networks lie. A test can pass by luck on an intermittent fault, and a test can fail for a reason unrelated to what you were testing, such as a tool that blocks certain traffic. So you sometimes need to ask the same question twice, in two different ways.

29.3.3 PLAIN - a worked example

1. Suppose the list of possible causes for the reader's fault starts like this.

```
A  The Mac's network settings are wrong
B  The Wi-Fi link to the router is bad
C  The home router is broken or blocking
D  The ISP's access network is broken
E  The ISP blocks this destination
F  The path into Microsoft is broken
G  Microsoft drops traffic from this address range
H  github.com is down
I  DNS gave a wrong or stale address
```

2. Nine causes. Now consider two candidate tests.
3. Test one: reload the page in a different browser. If it fails, you have removed nothing at all, because every cause above would still produce a failure. If it succeeds, you have learned that the first browser was at fault, which was not even on the list. Very low eliminating power.
4. Test two: from the same Mac, run a name lookup and then try to open a connection to the numeric address directly.
5. If the name lookup works and the connection still fails, you strike off I, because naming worked, and you learn the failure is later than naming.
6. Test three: load a different website, say `example.com`, from the same Mac on the same Wi-Fi. If it loads, you strike off A, B, C and D in one move, because a working page proves the Mac, the Wi-Fi, the router and the access network all carry traffic to the wider internet.
7. That single test removed four of nine causes. That is high eliminating power, and it takes two seconds.
8. Test four: mobile data. It strikes off H, because the site answered someone.
9. After those tests the list is E, F and G. Three causes instead of nine, in under a minute, and every step was justified.
10. Notice that no test so far can separate E, F and G, because all three produce identical silence from where you sit. That is an honest and useful place to stop and say so.

29.3.4 PLAIN – what is really happening inside

1. There is a hidden structure to good fault finding, and it is worth naming.
2. Every fault has a chain of things that must all work. Cable, address, router, route, naming, connection, encryption, application.
3. A chain fails at exactly one link first. Your job is to find the first broken link, not to admire the whole chain.
4. Searching a chain has a best method: cut it in the middle. Test something halfway along. Then you know which half holds the break, and you repeat.
5. That is called bisection, and it is the same method used to find which change in a code history broke a program.
6. The ladder in the next section is a chain laid out in order, from the nearest link to the furthest, so that you can walk it or bisect it.
7. Walking from the bottom is slower but never confuses you. Bisecting is faster but needs you to know the chain well.
8. One more rule. When a test surprises you, believe the test and change your model. Do not explain the result away.
9. Most long debugging sessions are long because somebody decided early what the answer was and then read every result as support for it.
10. Write down, in words, what you have proved after each step. A three-line note stops you from re-testing the same thing four times.

29.3.5 TECHNICAL – the engineer's version

1. Formally, define the hypothesis space H as the set of candidate root causes consistent with all evidence so far.
2. A test T partitions H into the subset consistent with a pass and the subset consistent with a fail. Define the eliminating power of T as the size of the smaller of those two subsets, expressed as a fraction of H.
3. A test with eliminating power near 0.5 is a bisecting test. A test with power near 0 is decorative.
4. This is the information-theoretic view: the ideal test maximizes expected information gain, which for a binary test is maximized when the two outcomes are equally likely.
5. Medicine names the same two properties. A highly sensitive test with a negative result rules a condition out. A highly specific test with a positive result rules it in. Network tests have exactly these two flavours.
6. Worked eliminating power for common tests, taking the nine-cause list above:

Test	Rules out on pass	Power
Different browser	Nothing	Very low
Load example.com	Local chain, 4 of 9	High
dig github.com	DNS, 1 of 9	Medium
Mobile data	Site down, 1 of 9	Medium
nc to port 443	Nothing new on fail	High on pass
Different port, 22	Port policy	High

7. Design rules that follow from this.
8. Hold the destination fixed and vary the source, or hold the source fixed and vary the destination. Never vary both.
9. Vary one layer at a time. Name resolution, then network reachability, then transport, then application.
10. Prefer tests whose failure mode is distinguishable. A silent timeout is a weak signal because many causes produce it. A TCP reset, an ICMP administratively prohibited message, or an HTTP 403 are strong signals because few causes produce them.
11. Record the exact command, the exact output and the timestamp. Use `date -u +%FT%TZ` on macOS to stamp a log line in coordinated universal time, so that evidence from two machines can be lined up.

12. Beware of the tool as a variable. `curl`, a browser, and `git` do not use identical name resolution, proxy settings or TLS libraries. A browser may use its own encrypted DNS while `curl` uses the system resolver.
13. Beware of caching as a hidden variable. Negative DNS answers, TCP connection reuse and HTTP caches can make a second test not independent of the first. Start a fresh process for each test where it matters.
14. State conclusions in the form: evidence, therefore eliminated set. Anything else is a hypothesis and must be labelled as one.

29.3.6 WORDS - remember these

1. Controlled experiment — change one thing only — a test in which a single independent variable is varied while all others are held constant.
2. Eliminating power — how much of your list a test removes — this book's name for the fraction of the hypothesis space excluded by either outcome.
3. Bisection — cut the chain in the middle — binary search over an ordered sequence of dependencies to locate the first failing link.
4. Confounding variable — something else that moved too — an uncontrolled variable that could equally explain the observed change.
5. Silent failure — nothing comes back at all — a drop with no ICMP error and no TCP reset, which is the least informative failure mode there is.
6. Strong signal — a failure that names its own cause — an explicit error such as TCP RST, ICMP type 3 code 13, or an HTTP status code.

29.4 The diagnostic ladder, rung by rung

29.4.1 PLAIN - in simple words

1. Getting a web page involves a chain of things that must all work.
2. The ladder puts that chain in order, from the closest thing to you up to the furthest thing away.
3. Rung 1: does my machine have a working network connection with an address.
4. Rung 2: can I talk to my own router, the box in my home.
5. Rung 3: can I talk to a machine out on the internet using its number.
6. Rung 4: can I turn a name into a number.
7. Rung 5: can I open a connection to the exact service I want, on its port.
8. Rung 6: does the full request, with encryption, actually succeed.
9. Rung 7: if it fails, how far along the path do my packets get.
10. You climb from 1 upwards. The first rung that fails is where to look.
11. Every rung passing means the rungs below it are fine. That is the whole value of putting them in order.
12. And every rung has a limit. Passing a rung proves something specific and nothing more, and it is worth knowing exactly what.

29.4.2 PLAIN - a picture in your head

1. Think of posting a parcel, and the checks a careful person makes.
2. Rung 1 is "do I have a pen and paper and a return address". Without an address of your own, nothing can come back.
3. Rung 2 is "is the postbox at the end of my street working". If your own postbox has been sealed, nothing leaves.
4. Rung 3 is "can I send anything at all to another city". Not to a specific person. Just anything, anywhere far away.
5. Rung 4 is "can I look up my friend's street address from their name".
6. Rung 5 is "is there a door at that address, and does it open".

7. Rung 6 is “did the person inside take the parcel, check my identity papers and give me a receipt”.
8. Rung 7 is “if it never arrived, which sorting office was the last to scan it”.
9. Notice you cannot sensibly ask question 6 before question 5. There is no point asking whether the person signed for it if the door does not exist.
10. That ordering is what stops you wasting time. Most people start at rung 6, fail, and then guess.

Where this comparison breaks: with a parcel, every stage leaves a record and somebody can tell you where it stopped. On a network the middle stages usually leave no record at all, and a packet can be dropped by a machine that never tells anyone, including its own operator.

29.4.3 PLAIN - a worked example

1. Here is the ladder as a table. macOS commands, since that is what the reader uses. Linux differences come later in this section.

Rung	Question	macOS command
1	Do I have an address	`ipconfig getifaddr en0`
2	Can I reach my router	`ping -c 3 192.168.0.1`
3	Can I reach the internet	`ping -c 3 1.1.1.1`
4	Can I resolve a name	`dig +short github.com`
5	Can I open the port	`nc -vz github.com 443`
6	Does the request work	`curl -sSI https://github.com`
7	Where does it stop	`traceroute -n github.com`

2. And here is what each rung settles, and what it leaves open.

Rung	A pass proves	A pass does not prove
1	Link up, address held	Anything can be reached
2	Local link works	Anything beyond the router
3	Routing off the LAN	Naming works
4	Naming works	The address is reachable
5	TCP reaches service	TLS or HTTP work
6	Whole stack works	It works for others
7	Path up to last hop	Silence means fault

3. Third table: what a failure at each rung means.

Rung	A failure means
1	No link, no DHCP, or wrong port
2	Wi-Fi, cable, or router fault
3	No route, or ICMP filtered
4	Resolver or DNS path problem
5	Filtering, route, or port shut
6	TLS, certificate, or server error
7	Only shows where replies stop

4. Two of those entries deserve a warning right away.
5. Rung 3 failing may only mean that ping replies are blocked, not that routing is broken. Always confirm with a second test that does not use ping.

6. Rung 7 failing to reach the end is the weakest evidence on the whole ladder. Silence at the end of a trace is completely normal.

29.4.4 PLAIN - what is really happening inside

1. Rung 1. Your machine needs three things: a working link, an address of its own, and a default route, which is the instruction "send anything not local to the router".
2. A pass proves the network card is on and the machine has an address, usually handed out by the router. A pass does not prove that any packet has ever left the building.
3. A failure means the cable is out, the Wi-Fi did not join, or the address handout failed. If you see an address starting `169.254`, the handout failed and the machine gave itself a placeholder.
4. Rung 2. Ping your own router. On the reader's network that is `192.168.0.1`.
5. A pass proves the wireless or cable link works, your address is on the right network, and the router is alive and answering.
6. A pass does not prove the router can reach anything else. A router with a dead internet line still answers you cheerfully.
7. A failure means the local link is broken, or your address is on a different network from the router, or the router blocks ping to itself, which is rare for home routers and common for company ones.
8. Rung 3. Ping a numbered address out on the internet, such as `1.1.1.1` or `8.8.8.8`. Use the number, never a name.
9. A pass proves your packets leave your network, cross your provider, reach a distant machine, and come back. That is a lot, and it is proved in one line.
10. It also separates two questions that beginners mix up: reaching things, and naming things. Using a number takes naming out of the experiment.
11. A pass does not prove that naming works, and does not prove any particular site is reachable. Different destinations take different paths.
12. A failure is ambiguous, because many networks drop ping on purpose. Retest with something that is not ping before concluding anything.
13. Rung 4. Turn a name into a number. This is the naming system doing its job.
14. A pass proves your machine can reach a name server and get an answer, so both naming and the path to the name server are healthy.
15. A pass does not prove the returned address is reachable, or correct, or fresh. A cached wrong answer looks exactly like a right one.
16. A failure means your name server is unreachable or refusing. Test again naming a different server directly to separate the two.
17. Rung 5. Open a connection to the actual port. Web pages over encryption use port 443. This is the first rung that tests the service you actually want.
18. A pass proves that packets reach the service, that the service is listening and that a full handshake completed in both directions. This is the strongest single proof of reachability there is.
19. A pass does not prove the encryption will succeed, that the certificate is valid, or that the site will answer with a page.
20. A failure with a refusal message means something answered and said no. A failure with silence means nothing answered at all, which is the harder case and the one the reader hit.
21. Rung 6. Make the real request. Now encryption and the web protocol are involved as well.
22. A pass proves everything below, plus certificate checking, plus a server that produced a real answer.
23. A pass does not prove it works for other people, other names on the same machine, or the rest of the site.
24. A failure here, with rung 5 passing, is a very specific and useful state: the network is fine and the problem is encryption or the application.
25. Rung 7. Find out how far your packets get by sending probes with a short life and collecting the complaints from each machine along the way.
26. A pass, meaning a complete trace, proves the forward path exists and each machine on it was willing

to identify itself.

27. A pass does not prove your actual traffic follows that same path, because the probe may use a different protocol and port from your real connection.
28. A failure, meaning stars at the end, mostly means the last machines choose not to reply. It is a hint, not a verdict.

29.4.5 TECHNICAL - the engineer's version

1. Rung 1, interface and address. Three commands, each answering a different part of the question.

```
ifconfig en0
ipconfig getifaddr en0
networksetup -getinfo "Wi-Fi"
netstat -rn -f inet | head -5
```

2. `ifconfig en0` prints link state and addresses. Look for `status: active` and an `inet` line. macOS prints the mask in hexadecimal, for example `netmask 0xffffff00`, which is a /24.
3. `ipconfig getifaddr en0` prints only the IPv4 address and exits with status 1 and no output if there is none, which makes it ideal inside scripts.
4. `networksetup -getinfo "Wi-Fi"` prints the service-level view: address, mask, router and hardware address. Use `networksetup -listallhardwareports` first to map `en0` to a service name, because on many Macs `en0` is Wi-Fi and on others it is a wired or adapter port.
5. `netstat -rn -f inet` shows the routing table. The `default` line names the gateway and interface. Without a default route, nothing off-link works.
6. On macOS, `utun` interfaces are point-to-point tunnels. The reader had several. They are created by VPN clients, by Apple's own services such as Back to My Mac and iCloud Private Relay, and by content filters. A `utun` with a default route can silently capture traffic. Check with `route -n get default` which interface actually wins.
7. Rung 1 pass criteria: `status: active`, a non-`169.254` address, a default route. `169.254.0.0/16` is link-local, defined in RFC 3927 from May 2005, and self-assigned when DHCP fails.
8. Rung 2, the default gateway. The reader's is `192.168.0.1`, inside the `192.168.0.0/16` private range from RFC 1918, published February 1996.

```
ping -c 3 -W 1000 192.168.0.1
arp -a -n | grep 192.168.0.1
```

9. `-c 3` sends three packets. `-W 1000` waits at most 1000 milliseconds per reply. Note that on macOS `-W` is milliseconds, while on Linux `-W` is seconds. This is a real and frequent source of confusion.
10. `arp -a -n` shows the Address Resolution Protocol cache. If the gateway appears with a hardware address, layer 2 resolution succeeded even if ICMP is filtered. An `(incomplete)` entry means the router did not answer ARP at all, which is a much more serious finding.
11. Rung 3, off-network reachability by address.

```
ping -c 5 1.1.1.1
ping -c 5 8.8.8.8
nc -vz -G 3 1.1.1.1 443
```

12. `1.1.1.1` is Cloudflare's resolver, launched 1 April 2018. `8.8.8.8` is Google Public DNS, launched 3 December 2009. Both answer ICMP echo and both listen on TCP 443, so you can test the same target two ways.
13. If ping fails and `nc` succeeds, ICMP is filtered and routing is fine. That pair of results is worth more than either alone.
14. Rung 4, name resolution.

```
dig +short github.com
dig @1.1.1.1 +short github.com
```

```
dig @8.8.8.8 +short github.com
scutil --dns | head -20
dscacheutil -q host -a name github.com
```

15. Plain `dig` uses the system resolver list. `dig @server` bypasses it and asks a named server directly. Running both separates "resolution is broken" from "my configured resolver is broken".
16. `scutil --dns` is the macOS-specific view of resolver configuration, including per-domain resolvers installed by VPN clients. This is where a `utun` interface reveals that it has hijacked name resolution for some domains.
17. `dscacheutil -q host -a name github.com` asks the macOS directory service, which is what applications actually use. `dig` talks to DNS directly and bypasses the system cache, so the two can disagree. That disagreement is itself a diagnosis.
18. Apple ships `dig`, `host` and `nslookup` as an old bundled copy of the BIND utilities from the Internet Systems Consortium. The bundled version has lagged upstream by years. If your Mac lacks `dig`, install a current one with `brew install bind` and use `dscacheutil` or `dns-sd -G v4 name` in the meantime. Treat the exact presence of these tools as version-dependent rather than guaranteed.
19. Rung 5, transport reachability.

```
nc -vz -G 5 github.com 443
nc -vz -G 5 20.207.73.82 443
curl -v --connect-timeout 5 -o /dev/null https://github.com
```

20. `-v` makes `nc` report the outcome, `-z` means connect and send no data, `-G 5` sets a five second TCP connection timeout. Without `-G`, `nc` waits for the operating system default.
21. On macOS the operating system default for a connection attempt is governed by `sysctl net.inet.tcp.keepinit`, whose default is 75000, in milliseconds. So an unanswered connection takes about 75 seconds to fail if no shorter timeout is set.
22. On Linux the equivalent is `net.ipv4.tcp_syn_retries`, default 6, giving roughly 127 seconds with exponential backoff from an initial retransmission timeout of 1 second as specified in RFC 6298 from June 2011.
23. Distinguish the three transport outcomes precisely:

Outcome	On the wire	Meaning
Connected	SYN, SYN-ACK, ACK	Service reachable
Refused	SYN, then RST	Host up, port shut
Timed out	SYN, silence	Dropped somewhere
Unreachable	ICMP type 3	Router said no

24. Rung 6, application level.

```
curl -sSI https://github.com
curl -sS -o /dev/null -w '%{http_code} %{time_total}\n' \
     https://github.com
```

25. `-I` sends a HEAD request, which asks for headers only. `-s` silences the progress meter, `-S` keeps error messages visible. Together `-sS` is the standard pairing for scripts.
26. A pass here exercises TCP, TLS 1.2 or 1.3, certificate chain validation against the system trust store, ALPN negotiation and HTTP.
27. Rung 7, path discovery.

```
traceroute -n -q 1 -w 2 -m 20 github.com
sudo traceroute -I -n github.com
sudo traceroute -n -P TCP -p 443 github.com
```

28. `-n` skips reverse name lookups and makes the trace much faster. `-q 1` sends one probe per hop instead of three. `-w 2` waits two seconds instead of five. `-m 20` stops at twenty hops instead of the macOS default, which is the system TTL, normally 64.
29. macOS `traceroute` defaults to UDP probes to ports from 33434 upwards, the base port chosen by Van Jacobson when he wrote the tool in 1987 at Lawrence Berkeley Laboratory.
30. Path asymmetry rule: traceroute shows the forward path only, one hop at a time, and each displayed round-trip time includes the return path from that intermediate router, which may be entirely different.

29.4.6 WORDS - remember these

1. Default route — where to send anything not local — the `0.0.0.0/0` entry in the routing table, pointing at the gateway.
2. Link-local address — the placeholder a machine gives itself — `169.254/16` under RFC 3927, a sign that DHCP failed.
3. ARP cache — the table of neighbours' hardware addresses — the result of Address Resolution Protocol, viewable with `arp -a`.
4. Three-way handshake — the opening of a connection — SYN, SYN-ACK, ACK, after which TCP considers the connection established.
5. RST — an explicit refusal — a TCP reset segment, meaning a host exists but nothing is listening or a device rejected the attempt.
6. HEAD request — ask for headers, not the page — an HTTP method returning the same headers as GET with no message body.
7. Bisection point — the rung that halves the problem — in practice rung 5, because it separates all network causes from all application causes.

29.5 The reader's own fault, mapped onto the ladder

29.5.1 PLAIN - in simple words

1. Now we take the real fault from the reader's session and put every piece of evidence on a rung.
2. The Mac had an address and a working local network. Rung 1 passed.
3. The router at `192.168.0.1` answered. Rung 2 passed.
4. Packets reached machines out on the internet and came back. Rung 3 passed.
5. The name `github.com` turned into the number `20.207.73.82` quickly and correctly. Rung 4 passed.
6. The attempt to open a connection to `20.207.73.82` on port 443 got nothing at all. Rung 5 failed, and failed by silence.
7. Because rung 5 failed, rung 6 never happened. There was no encryption step and no web request, because there was no connection to run them over.
8. The trace showed packets travelling through the reader's provider, into Microsoft's network at Delhi, through Mumbai, into Pune, and then silence from hop 13 onwards. Rung 7 gave a partial answer.
9. And the same site loaded instantly on mobile data, which killed the theory that GitHub was down.
10. Put those together and something very specific appears.
11. Naming worked but connecting did not. That combination is one of the most informative results you can get, and the next block explains why.

29.5.2 PLAIN - a picture in your head

1. Go back to the parcel. You looked up your friend's street address from a directory, and the directory answered instantly with a correct address.
2. So the directory service is fine, the roads to the directory office are fine, and you hold a real address in your hand.
3. You then walk to that address, knock, and nothing happens. No answer. No "go away". No locked-gate sign. Just silence.

4. And you know from a friend in another city that somebody was home the whole time and answered their knock.
5. What is left? Something between your street and that door swallowed you.
6. The important part of the picture is the silence. A locked gate is information. A sign saying "no entry" is information. Nothing at all is the deliberate absence of information.
7. Anything that wants to hide the fact that it is stopping you produces exactly this: nothing.

Where this comparison breaks: a person walking can see the buildings they pass. Your packet cannot. It has no memory of where it has been, and if it is dropped it does not scream. Neither end is told. That is not a flaw in your test, it is how the network was designed.

29.5.3 PLAIN – a worked example

1. Here is the reader's evidence, rung by rung.

Rung	Result	Conclusion
1 Interface	Pass	Mac has address, link up
2 Router	Pass	Local network is healthy
3 Public IP	Pass	Routing off the LAN works
4 DNS	Pass	Naming is not the fault
5 TCP 443	Fail, silent	Fault is at or before service
6 TLS, HTTP	Not reached	No information
7 Traceroute	Partial	Reaches Pune, then quiet

2. And the cross-check, which sits outside the ladder.

Cross-check	Result	Conclusion
Mobile data, same phone	Pass	Site is up, not down

3. A note on rung 3, told honestly. The session record does not include a `ping 1.1.1.1` result. What it does include is that name lookups against `1.1.1.1` worked.
4. That is stronger evidence than ping, not weaker. It means packets left the Mac, crossed the provider, reached Cloudflare's resolver, were processed, and the answers came all the way back.
5. So rung 3 is proved, but by a different observation than the one in the table. Say which observation proved a thing, not just that it is proved.
6. Here is the same picture as a diagram.

```
RUNG                        READER'S RESULT
 7  traceroute             |  hop 12 Pune, then * * *
 6  curl -sSI https://     |  never ran
 5  nc / curl to :443      |  FAIL - silence, 15 s
 4  dig github.com         |  PASS -> 20.207.73.82
 3  reach a public IP      |  PASS via DNS to 1.1.1.1
 2  ping 192.168.0.1       |  PASS - hop 1 answered
 1  ifconfig / address     |  PASS - address held
      ----------------------------------------
      break is between rung 4 and rung 5
```

29.5.4 PLAIN – what is really happening inside

1. Why is "naming works, connecting fails" such a strong pair of results?
2. Because the two use different machines, different protocols, different ports, and, most importantly, different destinations.

3. The name lookup went to `1.1.1.1`, a Cloudflare machine, over the simple message protocol on port 53.
4. The connection attempt went to `20.207.73.82`, a Microsoft machine, over the connection protocol on port 443.
5. Naming working proves the general machinery is fine: the Mac, the router, the provider's access network, the provider's core, and the ability to send and receive from a distant network.
6. So the fault cannot be anything general. It must be specific to something about the second attempt.
7. Only three kinds of thing differ between the two attempts: the destination address, the port and protocol, and the path taken to that destination.
8. That is the entire surviving list. Everything else has been eliminated by the naming test.
9. Now add the silence. The failure produced no refusal message and no error report from any router.
10. If the far machine were up but not listening, you would get a refusal. You did not.
11. If a router had no route, it would normally send back a "cannot get there" complaint. You did not get one.
12. If a firewall were configured to reject politely, it would say so. It did not.
13. Silence is what a device does when it is configured to drop, or when a packet enters a hole in the routing and dies without anyone taking responsibility.
14. Now add the mobile data result. The destination answered somebody else at the same time.
15. The honest conclusion: the destination service was up, the reader's own machine and local network were fine, naming was fine, and packets to that specific address on that specific port were being dropped somewhere between the reader's provider and that service, with no error returned.
16. What we cannot say: which device dropped them, whether it was intentional, or which organization owns it. The evidence does not reach that far.

29.5.5 TECHNICAL - the engineer's version

1. The evidence set, stated formally.
2. Layer 1 and 2: link active, ARP resolution to `192.168.0.1` succeeded, hop 1 in the trace answered. Eliminated as a cause.
3. Layer 3 local: address held, default route present, gateway responsive. Eliminated.
4. Layer 3 transit: UDP port 53 to `1.1.1.1` completed round trips with normal latency. Forward and return paths to Cloudflare's anycast prefix are intact. Provider access network and core eliminated as general failures.
5. Layer 4 naming path: `github.com` resolved to `20.207.73.82`, inside `20.192.0.0/10`, registered to Microsoft and announced by AS8075. Correct and geographically plausible for India. DNS eliminated.
6. Layer 4 transport to target: `curl -v https://github.com` printed `Trying 20.207.73.82:443...` and produced no further output for 15 seconds. No TCP RST, no ICMP type 3 of any code.
7. That means the SYN was sent and retransmitted according to the operating system's schedule, and nothing came back. On macOS a bare connect would run for about 75 seconds under `net.inet.tcp.keepinit`, so the 15 second cutoff indicates a client-imposed limit rather than the kernel giving up. Many tools set one.
8. Layer 7: never reached. No TLS ClientHello was ever sent, because TCP never established. Therefore no conclusion about certificates, SNI filtering at the TLS layer, or HTTP is available from this evidence.
9. Path evidence. The recorded trace, exactly as captured:

```
 1  192.168.0.1
 2  172.31.0.17
 3  137.97.29.249
 4  172.26.22.235
 5  172.16.18.33 / 172.26.14.75 / 172.26.22.234
 6  172.26.14.75 / 172.16.18.33
 7  ae66-0.del01-96cbe-1b.ntwk.msn.net (104.44.196.187)
 8  be23.rwa02.bom01.ntwk.msn.net (104.44.55.163)
 9  be1010.owr02.bom01.ntwk.msn.net (104.44.53.119)
10  be5.ibr02.pnq21.ntwk.msn.net (104.44.31.62)
```

```
11  ae104-0.rwa03.pnq21.ntwk.msn.net (104.44.20.52)
12  ae106-0.rwa04.pnq20.ntwk.msn.net (104.44.20.42)
13-20  * * *   (no reply)
```

10. Hops 2 to 6 are RFC 1918 private addresses inside the provider's core. `172.16.0.0/12` covers 172.16.x through 172.31.x, so `172.31.0.17`, `172.26.x` and `172.16.18.33` are all private. Hop 3, `137.97.29.249`, is public and belongs to the provider.
11. Hops with two or three addresses listed, namely 5, 6, 8, 9 and 11, are per-flow or per-packet load balancing across parallel links. Normal, not a fault.
12. From hop 7 the path is inside Microsoft's backbone, `ntwk.msn.net`. Site codes follow airport-style city codes: `del01` Delhi, `bom01` Mumbai, `pnq20` and `pnq21` Pune.
13. Router role codes `ibr`, `rwa` and `owr` mean inter-region backbone router, regional wide-area router and outer wide-area router. These are Microsoft naming conventions, not a standard. Interface prefixes `ae`, `be` and `po` all mean a bundle of physical links treated as one logical link, and are vendor-specific words for the same idea.
14. Hops 13 to 20 returning `* * *` proves nothing on its own. Control-plane ICMP time-exceeded generation is routinely rate-limited or disabled inside large backbones. Cisco IOS, for example, rate-limits ICMP unreachables to one every 500 milliseconds by default.
15. The decisive evidence is therefore not the trace. It is the pairing of a silent TCP timeout with a successful request from a different vantage point at the same time.
16. Surviving hypothesis set, in the reader's case:

Hypothesis	Consistent with evidence
Site down	No, refuted by mobile test
DNS wrong	No, address is correct
Local network fault	No, DNS round trips worked
Path or policy drop	Yes
Return path missing	Yes
Source prefix filtered	Yes
Transient routing event	Yes

17. Four hypotheses survive and no test performed so far separates them, because all four produce identical silence at the client.
18. Separating them requires evidence from outside the client: a TCP traceroute to port 443, a test to a second address in the same Microsoft prefix, a test from another host inside the same provider, or a looking-glass query from the destination side.

29.5.6 WORDS - remember these

1. Silent drop — nothing comes back — a packet discarded without generating ICMP or TCP error, by policy or by a routing black hole.
2. Black hole — a place packets vanish — a route that accepts traffic and discards it, often a null route or a stale advertisement.
3. Return path failure — your request arrives, the answer cannot come home — asymmetry where the destination network lacks a working route to your prefix.
4. Load-balanced hop — several addresses on one trace line — parallel links between the same two routers, chosen per packet or per flow.
5. Control plane — the router's own brain — the processor that generates ICMP errors and runs routing protocols, deliberately rate-limited.
6. Surviving hypothesis set — what is still possible — the causes not yet eliminated by any observation, which is the only honest output of a diagnosis.

29.6 Cross-checks that multiply your certainty

29.6.1 PLAIN - in simple words

1. A cross-check is a test that deliberately changes one big thing and keeps the rest as similar as possible.
2. Five of them cover almost every case, and each one eliminates a different family of causes.
3. Same destination, different network. Use mobile data or a hotspot.
4. Same network, different device. Use a phone on the same Wi-Fi, or a second computer.
5. Same destination, tested from far away. Use a shell on a server elsewhere, or a website that checks a site for you.
6. Same network, different destination. Load a different site on the same Wi-Fi.
7. Same destination, but tunnelled. Turn on a virtual private network, which wraps your traffic and sends it out from somewhere else.
8. Each of these answers a different question, and it is worth learning which.
9. The order matters too. Do the cheap ones first. Loading a different site takes two seconds and removes four causes.

29.6.2 PLAIN - a picture in your head

1. Think of a doctor with five simple tests and no scanner.
2. Press here. Does it hurt? Press there. Does it hurt? Each answer removes a set of possibilities.
3. No single press gives a diagnosis. The pattern of yes and no does.
4. The doctor is not looking for the answer in one test. The doctor is narrowing.
5. And a good doctor knows which press is worth doing first: the one that will change what they do next, whichever way it goes.
6. A test whose result would not change your next action is not worth doing.

Where this comparison breaks: a body is one system and stays put. Your network path can change between two presses, and a fault that lasts four minutes will give you a yes and then a no with no change in your hands.

29.6.3 PLAIN - a worked example

1. Here are the five cross-checks with exactly what each result eliminates.

Cross-check	Pass eliminates	Fail eliminates
Other network	Site down	Path-only causes
Other device, same net	That device	Device-specific causes
Public shell elsewhere	Site down	Site-side doubt
Other site, same net	Local chain	Destination-specific
Same site over VPN	Path policy	Nothing much

2. Walk through them one at a time with the reader's case.
3. Other network. The reader ran this. It passed. The claim "GitHub is down" is dead. What remains is everything on the reader's own path.
4. Other device, same network. If a phone on the same Wi-Fi also fails, the Mac is not the problem, and every Mac-specific idea can be struck off: the `utun` tunnel interfaces, a local firewall, a broken resolver setting.
5. If the phone succeeds on the same Wi-Fi, that is enormous. It means the router and the provider carry the traffic, and something on the Mac is the cause.
6. Public shell elsewhere. Any server you can log into, in any country, confirms the site independently and with a proper command rather than a phone browser.

7. Other site, same network. Loading `example.com` from the same Mac on the same Wi-Fi proves the Mac, the Wi-Fi, the router, the provider's access network and the provider's core all work.
8. That is the highest-value two-second test on the list.
9. Same site over a virtual private network. Your traffic is wrapped up and sent out from a machine elsewhere, so it enters the destination from a different address and a different path.
10. If it works over the tunnel and not without it, that is a strong pointer towards path or policy rather than anything at the destination.
11. But notice the honest entry in the table. A failure over the tunnel eliminates very little, because tunnels themselves can be blocked, slow, or misconfigured, and you have added a whole new set of possible faults.

29.6.4 PLAIN - what is really happening inside

1. Each cross-check works by holding a set of components constant and swapping one whole layer.
2. Different network swaps the entire path and your address, and keeps the destination.
3. Different device swaps the operating system, the software and the local settings, and keeps the path.
4. Different destination swaps the far end and most of the path, and keeps everything local.
5. Tunnel swaps the middle of the path and your apparent address, and keeps both endpoints.
6. Elsewhere-shell swaps everything except the destination.
7. Put them in a grid and you can see that between them they cover every component: your machine, your local network, your provider, the path, and the destination.
8. This is why five tests can produce a confident answer where one test cannot.
9. There is a trap here. Tests are only independent if they really do differ.
10. A phone on the same Wi-Fi that has fallen back to mobile data without you noticing is not a same-network test. Turn mobile data off first.
11. A second computer that shares your virtual private network is not an independent path.
12. And an online checker that tests from a data centre may reach a completely different front-door machine than a home connection would, so a pass there does not prove your address family or your region is served.

29.6.5 TECHNICAL - the engineer's version

1. Precise commands for each cross-check on macOS.

```
# 1. Same network, different destination
curl -sSI --max-time 8 https://example.com
curl -sSI --max-time 8 https://www.wikipedia.org

# 2. Same destination, force the address, skip DNS
curl -v --max-time 8 --resolve github.com:443:20.207.73.82 \
     https://github.com

# 3. Same destination, different port
nc -vz -G 5 github.com 22
nc -vz -G 5 github.com 443

# 4. Same destination, different address family
curl -4 -sSI --max-time 8 https://github.com
curl -6 -sSI --max-time 8 https://github.com
```

2. Test 2 is the one people forget. `--resolve host:port:address` tells `curl` to use that address for that name without a lookup, so you can test a specific front end while keeping the correct Server Name Indication and certificate name.
3. Test 3 separates destination reachability from port policy. In the reader's session, the git transports gave exactly this kind of split evidence: one transport worked while the other did not, and those two use different ports.

4. Test 4 matters because the session reported `IPv6: (none)`. If a destination is reachable over IPv6 and not IPv4 on your network, or the reverse, forcing the family names the fault immediately.
5. Independent vantage points, ranked by strength:

Vantage point	Independence	Effort
Mobile hotspot	High	Seconds
Friend on other ISP	Very high	Minutes
Cloud shell or VPS	Very high	Minutes
RIPE Atlas measurement	Highest	Hours
Web outage checker	Low	Seconds
Same-network device	Low path	Seconds

6. A looking glass is a public web or telnet interface on a provider's router that lets you run ping and traceroute from that provider's network towards you. Many transit providers and internet exchanges publish one. This is the only easy way to see the return path towards your own address.
7. RIPE Atlas, run by the RIPE Network Coordination Centre since 2010, will run a traceroute or TLS check towards your target from probes in chosen countries or chosen autonomous systems, which turns "is it just me" into a measurable question.
8. Record for each cross-check: source address family, source autonomous system, resolver used, resolved address, port, exact command, exact result, and a coordinated universal time stamp. Without the resolved address, two tests may not have been aimed at the same machine at all.

29.6.6 WORDS - remember these

1. Cross-check — a test that swaps one whole layer — an independent measurement designed to eliminate a family of causes at once.
2. Vantage point — where you measure from — a source with its own address, provider and path to the target.
3. Looking glass — someone else's router you may query — a public interface for running ping, traceroute and BGP queries from a remote network.
4. Server Name Indication — telling the server which site you want — the SNI field in the TLS ClientHello, sent before encryption of the certificate exchange, and therefore visible to middleboxes.
5. Independent test — one that shares nothing with the last — a measurement whose failure modes do not overlap with the previous one.
6. Address family — the kind of address in use — IPv4 or IPv6, selectable in most tools with `-4` and `-6`.

29.7 Ping, honestly

29.7.1 PLAIN - in simple words

1. Ping sends a tiny message that means "are you there", and waits for a message back that means "yes".
2. That is the whole of it. It does not fetch a page. It does not check a service. It does not know what a website is.
3. It is not part of the web, and it is not part of the connection machinery that carries your data. It is a separate, simpler kind of message.
4. Because it is separate, a machine can answer ping while its website is broken, and can refuse ping while its website works perfectly.
5. Both of those situations are completely normal. Neither is a bug.
6. Ping is the most over-trusted tool in networking. People treat "it pings" as proof that a service works, and "it does not ping" as proof of a fault.
7. Neither reading is safe. Ping tells you about one particular kind of message reaching one particular machine and coming back.

8. It is still useful. It is fast, it needs no name lookup if you use a number, and it measures how long the round trip takes.
9. Use it to measure, not to conclude.

29.7.2 PLAIN - a picture in your head

1. Ping is shouting “hello” at a building and listening for “hello” back.
2. If you hear it, someone is inside and the air between you carries sound.
3. If you hear nothing, there are three possibilities and you cannot tell them apart: nobody is in, somebody is in but has been told not to shout back, or the wind ate your shout.
4. And even a cheerful “hello” back tells you nothing about whether the shop inside is open, whether it has your item, or whether the till works.
5. The shout and the shopping are different activities.

Where this comparison breaks: your shout travels in a straight line, but a ping travels through a dozen machines that each decide whether to pass it on, and several of them treat it as the least important thing they handle that second.

29.7.3 PLAIN - a worked example

1. Here is real output from a Mac, with every field named.

```
$ ping -c 4 1.1.1.1
PING 1.1.1.1 (1.1.1.1): 56 data bytes
64 bytes from 1.1.1.1: icmp_seq=0 ttl=57 time=8.412 ms
64 bytes from 1.1.1.1: icmp_seq=1 ttl=57 time=9.021 ms
64 bytes from 1.1.1.1: icmp_seq=2 ttl=57 time=8.703 ms
64 bytes from 1.1.1.1: icmp_seq=3 ttl=57 time=9.633 ms

--- 1.1.1.1 ping statistics ---
4 packets transmitted, 4 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 8.412/8.942/9.633/0.441 ms
```

2. `56 data bytes` is what you sent. Add the 8-byte header and you get the `64 bytes` that came back.
3. `icmp_seq` is a counter so you can tell which reply belongs to which request, and see if any went missing. On macOS it starts at 0. On Linux it starts at 1.
4. `ttl=57` is the remaining life of the reply packet. Every packet starts with a number and every machine that forwards it subtracts one. When it hits zero the packet is thrown away, which is what stops packets circling forever.
5. You can read hop count from it. Most systems start replies at 64. Seeing 57 means about 7 machines forwarded it on the way back.
6. `time=8.412 ms` is the round trip: out and back, in thousandths of a second. It is not the one-way time and you cannot halve it reliably, because the two directions may take different routes.
7. `0.0% packet loss` is the fraction that never came back.
8. The last line is the spread. A small difference between minimum and maximum means a steady path. A large one means the path is congested or wireless is struggling.

29.7.4 PLAIN - what is really happening inside

1. Ping works but the site does not. Why this is normal.
2. The machine answering ping may not be the machine running the site. Large services put a shared address at the front, and the thing that answers a simple “are you there” is not the thing that serves pages.
3. The site’s software may have crashed while the machine itself is fine.
4. The port you need may be closed or filtered while the simple message type is allowed.
5. Ping fails but the site works. Why this is also normal.

6. Very many networks are configured to ignore this message type, on purpose, to reduce noise and to make scanning harder.
7. Routers treat it as low-priority work handled by their slow general-purpose processor, while real traffic goes through fast dedicated hardware. Under load, the simple message is dropped first.
8. Some large cloud platforms block it by default. On Amazon Web Services, a new security group does not permit it unless you add a rule. Traffic through several kinds of cloud load balancer does not answer it at all.
9. The practical rule: use ping to measure a path you already know works, and never use its failure alone as evidence.
10. And how to read loss. One or two percent on a wireless link is common and usually harmless for web browsing. Ten percent will make everything feel broken. A hundred percent means either a real break or a policy of silence, and you cannot tell which from ping.
11. The most misleading case: loss shown at a middle hop of a trace but not at the end. That is almost always the middle router deprioritizing replies to itself, not loss on the path. If the final destination shows no loss, the path has no loss.

29.7.5 TECHNICAL – the engineer's version

1. Ping uses the Internet Control Message Protocol, specified by Jon Postel in RFC 792 in September 1981. Echo Request is type 8, code 0. Echo Reply is type 0, code 0.
2. RFC 1122, the host requirements document from October 1989, states that every host must implement an ICMP Echo server. Blocking it is therefore a deliberate deviation from the standard, and a very common one.
3. Mike Muuss wrote the original `ping` in December 1983 at the United States Army Ballistic Research Laboratory, naming it after sonar.
4. The default payload is 56 bytes, making a 64-byte ICMP message and an 84-byte IPv4 packet including the 20-byte IP header.
5. The first 8 bytes of payload on BSD systems carry a timestamp, which is how the round-trip time is computed without keeping state.
6. Initial TTL values by system, useful for inferring hop counts:

System	Initial TTL
Linux, macOS, BSD	64
Windows	128
Cisco IOS, Solaris	255

7. ICMP messages you should recognize when they do appear:

Type and code	Meaning
0/0	Echo reply
3/0	Network unreachable
3/1	Host unreachable
3/3	Port unreachable
3/4	Fragmentation needed
3/13	Administratively prohibited
11/0	Time to live exceeded

8. Type 3 code 13 is a filter announcing itself. Type 3 code 4 is a path maximum transmission unit problem and causes connections that establish and then stall on the first large transfer.
9. Type 11 code 0 is the message traceroute depends on entirely.

10. Rate limiting is standard practice. Cisco IOS by default limits generation of ICMP unreachables to one every 500 milliseconds. Linux uses `net.ipv4.icmp_ratelimit`, default 1000 milliseconds, with a bitmask controlling which types are limited.
11. For IPv6 the message set is ICMPv6, RFC 4443 from March 2006, and filtering it wholesale breaks path MTU discovery and neighbour discovery. RFC 4890, May 2007, gives explicit guidance on which ICMPv6 types must be permitted.
12. Ping proves ICMP reachability of an address. It does not prove TCP or UDP reachability, service health, or that the responder is the service host, because anycast and load balancers separate the two.

29.7.6 WORDS - remember these

1. ICMP — the network's own message system — Internet Control Message Protocol, RFC 792, carrying errors and diagnostics rather than user data.
2. Echo request and reply — "are you there" and "yes" — ICMP types 8 and 0, what `ping` sends and receives.
3. TTL — how many machines a packet may pass — an 8-bit hop counter decremented by each router, discarded at zero, called hop limit in IPv6.
4. Round-trip time — out and back, in milliseconds — measured from a timestamp embedded in the echo payload.
5. Jitter — how much the timing wobbles — variation in round-trip time, shown as the standard deviation in ping's summary line.
6. Control plane policing — routers protecting their own brain — rate limits on traffic addressed to the router itself, the reason middle hops show loss.

29.8 The commands in full, for macOS

29.8.1 PLAIN - in simple words

1. A command is a word you type into the Terminal application, sometimes followed by short options that begin with a dash.
2. Those options are called flags. Each flag changes one small thing about how the command behaves.
3. You do not need to memorize them. You need to know which command answers which question, and about three flags for each.
4. The rest of this section is a reference. Read it once, then come back to it.
5. Everything here is for macOS. Where Linux differs, it says so, because the differences are exactly the kind that waste an afternoon.

29.8.2 PLAIN - a picture in your head

1. Think of a mechanic's toolbox with a dozen tools, not one big machine.
2. A torch shows you whether a thing is there. A meter measures. A stethoscope listens without touching anything.
3. Nobody uses a torch to measure voltage. Each tool answers one question.
4. `ping` is the torch. `curl` is the meter. `tcpdump` is the stethoscope.
5. Picking up the right one first is most of the skill.

Where this comparison breaks: mechanics' tools do not change what they are measuring. Some network tools do. A packet capture can slow a machine, and a port scan can trip a security system into blocking you.

29.8.3 PLAIN – a worked example

1. One question, five tools, five different answers.

```
ping -c 3 20.207.73.82        # does it answer hello
traceroute -n 20.207.73.82    # how far do probes get
nc -vz -G 5 20.207.73.82 443  # does the port open
curl -sSI https://github.com  # does the site answer
sudo tcpdump -n host 20.207.73.82   # what is on the wire
```

2. In the reader's case, the fourth and third both fail with silence, the second stops at hop 12, and the fifth would show outgoing SYN packets with nothing coming back at all.
3. That last observation, from the packet capture, is the one that converts "it does not work" into "we send, they never reply".

29.8.4 PLAIN – what is really happening inside

1. These tools split into four groups, and knowing the group tells you what a result is worth.
2. Group one asks the operating system what it thinks: `ifconfig`, `networksetup`, `scutil`, `netstat`, `route`, `arp`, `lsof`. They never send a packet. They report configuration and state.
3. Group two sends probes and interprets replies: `ping`, `traceroute`, `mtr`, `nmap`.
4. Group three does real work with a real protocol: `dig`, `nc`, `curl`.
5. Group four watches without taking part: `tcpdump`.
6. Group one can be wrong about reality. It shows intention, not outcome.
7. Group four cannot be wrong about what left your machine, which is why it settles arguments.
8. When groups three and four disagree with group one, believe four, then three, then two, then one.

29.8.5 TECHNICAL – the engineer's version

1. The command table. One line each on what the command proves.

Command	What it proves
`ifconfig`	Link state and addresses
`ipconfig getifaddr en0`	An IPv4 address is held
`ipconfig getpacket en0`	What DHCP actually gave
`networksetup`	Service-level configuration
`scutil --dns`	Which resolvers apply
`netstat -rn`	Routing table contents
`route get`	Which route a target uses
`arp -a`	Layer 2 neighbour resolution
`ping`	ICMP reachability only
`traceroute`	Where probe replies stop
`mtr`	Loss and latency over time
`dig`	DNS resolution, per server
`nslookup`	DNS, older interactive tool
`host`	DNS, short one-line answers
`nc -vz`	TCP or UDP port reachability
`curl -v`	Full TCP, TLS and HTTP path
`lsof -i`	Which process owns a socket
`tcpdump`	What actually left the machine
`nmap`	Port state across a range

2. `ping` flags that matter on macOS.

Flag	Meaning
`-c n`	Stop after n packets
`-i s`	Seconds between packets
`-s n`	Payload bytes, default 56
`-t s`	Give up after s seconds
`-W ms`	Per-reply wait, milliseconds
`-D`	Set the do-not-fragment bit
`-n`	No reverse name lookups

3. Linux difference, and it bites: on Linux `-t` sets the TTL, not a timeout; the deadline flag is `-w` in seconds, and `-W` is a per-reply timeout in seconds rather than milliseconds. Intervals below 0.002 seconds require superuser rights on macOS.
4. `traceroute` flags on macOS: `-I` uses ICMP echo instead of UDP; `-P proto` selects UDP, TCP, GRE or ICMP; `-p port` sets the base port, default 33434; `-n` prints numbers only; `-q n` sets probes per hop, default 3; `-w s` sets the wait, default 5 seconds; `-m n` sets maximum hops, defaulting to the system TTL of 64; `-f n` sets the starting hop.
5. Linux difference: the common Linux traceroute takes `-T` for TCP and `-U` for UDP, defaults to 30 hops, and ships alongside `tracepath`.
6. `dig` options: `+short` prints only the answer; `+trace` walks the hierarchy from the root itself; `@server` queries a named server directly; `-x addr` does a reverse lookup; `+tcp` forces TCP instead of UDP, which is how you test large answers and firewalls that only allow UDP port 53.
7. `nc` flags: `-v` reports the outcome, `-z` connects and sends no data, `-w s` sets an idle timeout, `-G s` sets the TCP connection timeout on macOS, `-u` uses UDP, `-n` skips lookups, `-4` and `-6` force a family.
8. UDP with `nc -uvz` is close to meaningless as a test, because a silent UDP port and an open one look identical. Only an ICMP port-unreachable reply distinguishes them, and that is often filtered.
9. `curl` flags: `-v` shows the connection steps; `-I` sends HEAD; `-s` silences the meter; `-S` keeps errors visible; `--resolve host:port:addr` pins an address without DNS; `--connect-timeout s` bounds the TCP phase; `--max-time s` bounds everything; `-w fmt` prints timing fields; `-4` and `-6` force an address family.
10. `curl` exit codes worth recognizing:

Code	Meaning
6	Could not resolve host
7	Failed to connect
28	Operation timed out
35	TLS handshake failure
52	Empty reply from server
60	Certificate not trusted

11. Code 6 is a rung 4 fault. Code 7 and 28 are rung 5. Codes 35 and 60 are rung 6. The exit code alone places the fault on the ladder.
12. Timing breakdown in one command:

```
curl -o /dev/null -sS -w \
'dns %{time_namelookup}\ntcp %{time_connect}\n\
tls %{time_appconnect}\nttfb %{time_starttransfer}\n\
total %{time_total}\nip %{remote_ip}\n' \
https://github.com
```

13. Each field is cumulative from the start, so subtract to get phase durations. A large gap between `tcp` and `tls` is a TLS problem; a large gap between `tls` and `ttfb` is a server-side problem.

14. macOS-specific state commands: `scutil --dns` shows the ordered resolver list including per-domain entries installed by VPN software; `ipconfig getpacket en0` prints the full DHCP reply with lease time, router and `domain_name_server`; `route -n get default` names the winning interface; `networksetup -listallhardwareports` maps `en0` to Wi-Fi or Ethernet, which differs between Mac models.
15. `tcpdump` needs `sudo`. Useful invocations:

```
sudo tcpdump -i en0 -n host 20.207.73.82
sudo tcpdump -i en0 -n port 443 and host 20.207.73.82
sudo tcpdump -i en0 -n -w /tmp/gh.pcap host 20.207.73.82
```

16. `-i` picks the interface, `-n` stops name lookups, `host` and `port` are filter expressions in the Berkeley Packet Filter language described by Steven McCanne and Van Jacobson at the USENIX conference in winter 1993, and `-w` writes a capture file you can send to somebody.
17. `mtr` and `nmap` are not shipped with macOS. Install with `brew install mtr nmap`. `mtr` needs `sudo` for ICMP mode; `mtr --tcp --port 443 github.com` probes with TCP instead. `nmap -Pn -p 443 --reason github.com` reports why it reached its conclusion, which is the single most useful nmap flag for diagnosis.
18. Linux equivalents, briefly: `ip addr` for `ifconfig`, `ip route` for `netstat -rn`, `ip neigh` for `arp -a`, `ip route get` for `route get`, `ss -tunap` for `lsof -i`, and `resolvectl status` for `scutil --dns` on systems using systemd-resolved.

29.8.6 WORDS - remember these

1. Flag — a short option after a command — a switch, usually one dash and a letter, that alters one behaviour.
2. HEAD request — headers only, no body — the HTTP method `curl -I` sends.
3. Exit code — the number a command leaves behind — readable with `echo $?`, and for `curl` it names the failing layer.
4. Berkeley Packet Filter — the language for choosing packets — the expression syntax used by `tcpdump`, from a 1993 USENIX paper.
5. Homebrew — the usual way to install extra tools on a Mac — a package manager providing `mtr`, `nmap`, `bind` and much else.
6. Reverse lookup — number to name — `dig -x` or `host`, using the `in-addr.arpa` tree, and frequently absent or wrong.

29.9 The decision tree for "I cannot reach a site"

29.9.1 PLAIN - in simple words

1. A decision tree is a list of steps where each answer sends you to a different next step.
2. The point is that you stop deciding what to do next. The tree decides.
3. That matters because fault finding goes wrong when you are annoyed and start guessing.
4. Follow it in order. Do not skip ahead because you have a hunch.
5. Every branch ends either in a cause, or in a clear instruction about who to contact and what to tell them.

29.9.2 PLAIN - a picture in your head

1. It is the flowchart on the wall of a repair shop.
2. Junior staff follow it and get the same answer a senior person would.
3. The chart is not clever. It is just written down, so nobody forgets step four when they are in a hurry.
4. Your own memory under pressure is the unreliable part, not your knowledge.

Where this comparison breaks: a repair chart assumes one machine with fixed parts. A network path is shared with strangers and changes shape between two steps, so a tree can send you down a branch that was correct thirty seconds ago and is wrong now. Re-run the step that surprised you before you trust it.

29.9.3 PLAIN - a worked example

1. The procedure, written to be followed without thinking.

```
 1 Load a different site (example.com).
     works -> go to 2
     fails -> go to 10
 2 dig +short github.com
     gives an address -> go to 3
     fails -> go to 12
 3 nc -vz -G 5 <that address> 443
     connects -> go to 4
     refused  -> service or port issue, go to 8
     silence  -> go to 5
 4 curl -sSI https://github.com
     works -> the fault was the browser or a cache
     TLS error -> certificate or clock, go to 9
 5 Test from mobile data or another network.
     works there -> path fault, go to 6
     fails there too -> go to 8
 6 Test another device on your own Wi-Fi.
     also fails -> not your Mac, go to 7
     works -> fault is on your Mac, check utun and firewall
 7 traceroute -n and TCP traceroute to port 443.
     record where it stops, then go to 11
 8 Check the service status page and an outage tracker.
     confirmed outage -> stop, wait, nothing to fix
     nothing reported -> go to 11
 9 Check date and time, then the certificate chain.
10 Restart Wi-Fi, then check rung 1 and rung 2 commands.
     still fails -> local network or router fault
11 Log the fault for an hour with the script in 29.12.
     then contact the ISP with the evidence in 29.10
12 dig @1.1.1.1 and dig @8.8.8.8 for the same name.
     one works -> your configured resolver is the fault
     both fail -> port 53 is blocked or routing is broken
```

29.9.4 PLAIN - what is really happening inside

1. Step 1 is first because it has the highest eliminating power for two seconds of effort. It clears your machine, your Wi-Fi, your router and your provider's access network in one go.
2. Step 2 is second because naming failures produce confusing symptoms everywhere else, so you remove them early.
3. Step 3 is the bisection point of the whole tree. Above it, everything is application. Below it, everything is network.
4. Step 3 has three outcomes, not two, and the difference between refused and silent is the single most valuable distinction in the tree.
5. Step 5 exists to close the "is it down" question before you spend an hour on your own network.
6. Steps 10 and 12 are the two dead ends that you can actually fix yourself.
7. Step 11 is the honest ending. Sometimes the correct output of a diagnosis is a well-evidenced report to somebody with access you do not have.

29.9.5 TECHNICAL - the engineer's version

1. Dead ends and the correct action at each.

Dead end	Action
Refused on 443	Retest by IP and by name
Silent on 443, mobile OK	Collect trace, report to ISP
Both resolvers fail	Test port 53 with +tcp
No default route	Renew DHCP, check utun
Works on other device	Audit local firewall, VPN
Status page confirms	Stop, no action available

2. Stop and contact the internet provider when all of the following hold: your local chain passes, another destination on the same network works, DNS resolves correctly, a connection to the target port times out with no reply, the same target succeeds from another provider, and the condition persists across at least twenty minutes of logging.
3. What to give them so it is not dismissed: your account or circuit identifier, your public address at the time from `curl -s ifconfig.me` or similar, the target address and port, a numeric traceroute, the exact `curl` or `nc` output including the timeout, a coordinated universal time stamp, and one sentence stating that the same target works from a different provider at the same moment.
4. Phrase the request as a specific technical question. "Please check for a routing or filtering issue between AS-of-yours and 20.207.73.82 on TCP 443" gets escalated. "The internet is not working" gets a modem reboot script.
5. Ask them explicitly to check the return path towards your prefix, not only the forward path. Return-path faults are invisible from your side and are a common cause of exactly this symptom.
6. If they claim it is a customer-side issue, the counter-evidence is the working test to a different destination on the same line, which proves your line carries traffic.

29.9.6 WORDS - remember these

1. Decision tree — a written list of if-this-then-that steps — a diagnostic procedure with defined branch conditions and terminal states.
2. Dead end — a branch where you personally can do no more — a terminal state whose action is escalation rather than repair.
3. Escalation — passing the fault to whoever can act — a handover accompanied by evidence sufficient to reproduce the finding.
4. Circuit identifier — the number your provider knows you by — the account or service identifier that lets a technician find your line.
5. Prefix — a block of addresses — a range written as an address plus a slash and a length, such as `20.192.0.0/10`.

29.10 Writing up a fault so someone else can act

29.10.1 PLAIN - in simple words

1. A report is not a complaint. It is a set of instructions for reproducing what you saw.
2. Four things make a report work: what you did, what you expected, what happened, and what you have already ruled out.
3. The last one is the one people leave out, and it is the one that saves the most time.
4. "The internet is broken" gives the reader nothing to check, so they send you the standard script: restart the router.
5. A traceroute plus a command output gives them something to paste into their own tools, so it goes to somebody technical.
6. Keep every line copy-and-pasteable. Never describe output in your own words when you could paste it.

29.10.2 PLAIN - a picture in your head

1. Two people go to a mechanic.
2. The first says "it makes a funny noise sometimes".
3. The second says "at 40 kilometres per hour, in third gear, on a left turn only, there is a knock from the front right, and it started on Tuesday".
4. The second person gets a fix. The first gets a bill for an hour of guessing.
5. The information in the second sentence is exactly the information a good fault report contains: conditions, location, timing, and what does not trigger it.

Where this comparison breaks: a mechanic can drive the car. Your reader cannot sit at your desk, so anything you did not write down does not exist.

29.10.3 PLAIN - a worked example

1. A template you can fill in.

```
Subject: TCP 443 to 20.207.73.82 times out from my line

WHAT I DID
  From macOS on my home connection, 13 Aug 2026, 09:40 UTC:
  curl -v --max-time 15 https://github.com

WHAT I EXPECTED
  A TCP connection, then an HTTP 200 response.

WHAT HAPPENED
  * Trying 20.207.73.82:443...
  curl: (28) Failed to connect after 15001 ms
  No RST, no ICMP unreachable. Total silence.

WHAT I HAVE RULED OUT
  - DNS: dig +short github.com -> 20.207.73.82, in 12 ms
  - Local network: ping 192.168.0.1 -> 0% loss
  - Internet access: https://example.com loads normally
  - Service down: same URL loads instantly on mobile data
  - My device: same failure from a second device on this Wi-Fi

EVIDENCE
  traceroute -n reaches 104.44.20.42 (hop 12) then * * *
  My public address at the time: <address>
  Circuit / account: <identifier>

WHAT I AM ASKING
  Please check for a routing or filtering problem towards
  20.207.73.82 on TCP 443, including the return path to my
  prefix. Happy to run any command you name.
```

29.10.4 PLAIN - what is really happening inside

1. Why this format works is not politeness. It is about who can act.
2. A first-line support agent has a script and a set of buttons. Your report must contain something their script recognizes, or something so specific that the script does not fit and they must pass it on.
3. A network engineer needs three things to look at their own tools: a source, a destination, and a time. Give all three or they cannot look.
4. The "ruled out" list is doing a specific job. It pre-empts the four questions they would otherwise ask you, each costing a day of email.

5. Pasting exact output matters because your summary may hide the detail that identifies the fault. "It did not work" and "no RST, no ICMP" are wildly different findings.
6. State facts and separate them from your theory. Put any theory in one line at the end, marked as a guess. Reports that assert a cause get argued with; reports that present evidence get investigated.

29.10.5 TECHNICAL - the engineer's version

1. Minimum evidence set for a reachability report:

Item	Why it is required
Source public address	Identifies your prefix
Destination address	Identifies the target
Port and protocol	Narrows any filter
UTC timestamp	Lets them read logs
Numeric traceroute	Shows last responding hop
curl or nc output	Shows the failure mode
Control test result	Proves the line works

2. Capture the public address at the time of the fault, since it may change: `curl -s https://ifconfig.me` or `dig +short myip.opendns.com @resolver1.opendns.com`.
3. Always paste numeric output. `traceroute -n` avoids reverse lookups that might themselves fail and confuse the reader.
4. Include one control test that succeeded. Without it, the first response will be a line test rather than a routing investigation.
5. Where a packet capture is appropriate, attach a file made with `sudo tcpdump -i en0 -n -w /tmp/gh.pcap host 20.207.73.82`. Retransmitted SYN packets with no response are unambiguous proof of a silent drop and are the strongest artefact you can supply.
6. Use coordinated universal time throughout and say so. Provider logs are kept in it, and a mismatch of five and a half hours makes your evidence unsearchable.
7. Keep the theory to one labelled line, for example: "Hypothesis, unproven: asymmetric routing or a policy filter between your network and AS8075."

29.10.6 WORDS - remember these

1. Reproduction steps — how to see it yourself — an exact command sequence that produces the reported behaviour.
2. Control test — the thing that worked — a comparison measurement proving the surrounding system is healthy.
3. Artefact — a file that carries evidence — a capture, a log or a saved output, attached rather than described.
4. First-line support — the scripted layer — the tier whose tools cannot see routing, whose job is to filter and escalate.
5. Coordinated universal time — the one clock everyone shares — the reference time zone used in logs, printed by `date -u`.

29.11 When it is not your problem

29.11.1 PLAIN - in simple words

1. Sometimes the fault really is at the other end, or really is your provider, and the right action is to stop working and wait.
2. There are three places to look, and each one has a weakness you must know.

3. A status page is written by the company that owns the service.
4. An outage tracker counts complaints from members of the public.
5. Your provider's own channels tell you about local work and known faults.
6. None of the three is proof. Together with your own tests, they are enough.

29.11.2 PLAIN - a picture in your head

1. A status page is the shop's own sign in the window saying "we are open".
2. An outage tracker is a crowd outside the shop, some of whom came on the wrong day and some of whom wanted a different shop.
3. Your own test is walking to the door and pulling the handle.
4. Only one of those three actually touches the door.

Where this comparison breaks: a shop sign is updated by someone who can see the shop. A status page is often updated by a process that requires a human decision and an internal agreement that an incident exists, which can lag the outage by half an hour or more.

29.11.3 PLAIN - a worked example

1. Three checks and what each is worth.

Source	Strength	Weakness
Official status page	Authoritative	Slow, coarse regions
Outage tracker	Fast, crowd-driven	Noisy, biased to big ISPs
Your own cross-check	Direct evidence	Only one vantage point

2. GitHub publishes a status page, hosted on Atlassian's Statuspage product, which Atlassian acquired in 2016. It reports per-component state such as Git Operations, API Requests, Webhooks, Actions and Pages.
3. That component split matters. In the reader's session plain git kept working while the web interface and the programming interface did not. A status page showing "Git Operations operational" and something else degraded would be consistent with that.
4. Downdetector, launched in 2012, plots reports per minute. A sharp spike across many providers means a service problem. A spike from one provider only means a provider problem.
5. If the tracker shows nothing and your cross-check from another network works, the problem is yours or your provider's. That is the common case.

29.11.4 PLAIN - what is really happening inside

1. How to tell a regional problem from a personal one, in three moves.
2. Move one: another destination on the same line. Works means your line is fine and the fault is destination-specific.
3. Move two: the same destination from another provider. Works means the destination is fine and the fault is provider-specific or path-specific.
4. Move three: ask one other person on the same provider in the same city. If they fail too, it is regional and the provider will already have a ticket.
5. Combine them into a small truth table and the answer falls out without argument.

Same line, other site	Other line, same site	Verdict
Works	Works	Path or policy fault
Works	Fails	Service problem
Fails	Works	Your line or router

Same line, other site	Other line, same site	Verdict
Fails	Fails	Wide outage

6. Two honest limits. A status page saying "all systems operational" never contradicts your own failed request, because the service may be fine and unreachable from you. And a tracker with no reports never proves an absence, because trackers under-report small providers and small countries.

29.11.5 TECHNICAL - the engineer's version

1. Verification independent of anybody's page:

```
curl -sS -o /dev/null -w '%{http_code} %{remote_ip}\n' \
     --max-time 10 https://www.githubstatus.com
dig +short api.github.com
curl -sSI --max-time 10 https://api.github.com | head -3
```

2. Query the status host from your own machine and note the resolved address; if the status page itself is unreachable from your network, that is another data point about your path, not about the service.
3. For provider-wide issues, check whether the provider's own portal and speed-test host respond, and whether a public looking glass in your country can reach your prefix.
4. Autonomous system level tools separate provider from path: `whois -h whois.cymru.com " -v 20.207.73.82"` returns the announcing autonomous system, which for Microsoft is AS8075. Doing the same for your own public address names your provider's autonomous system, and that pair is what an engineer needs to investigate.
5. Outage aggregation has known limits: crowd trackers measure complaints, not reachability, so they lag by minutes, over-represent consumer services and large providers, and cannot see business-hours-only faults.
6. Where experts disagree: some operators argue public status pages should be automated from probes, others that human confirmation prevents false alarms and customer panic. The practical consequence for you is that a page can be green during a real outage, so never treat it as a veto over your own measurements.

29.11.6 WORDS - remember these

1. Status page — the operator's own report — a published per-component state, updated by an internal incident process.
2. Outage tracker — a count of public complaints — a crowd-sourced signal that correlates with outages but does not measure them.
3. Regional fault — broken for a city or a provider — a failure whose blast radius is a geography or an autonomous system.
4. Truth table — every combination of results — a grid mapping test outcomes to conclusions, removing the need for judgement.
5. Whois — who owns this number — a directory service naming the holder of an address block or autonomous system number.

29.12 Intermittent faults, the hardest kind

29.12.1 PLAIN - in simple words

1. An intermittent fault is one that comes and goes.
2. It defeats every test in this chapter, because every test is one shot.
3. You run a command, it passes, and you conclude the problem is gone. Ten minutes later it is back.
4. The only cure is to stop taking single measurements and start recording, continuously, with the time written next to every result.

5. Then the pattern appears: every evening, every hour, only under load, only after the router has been up for two days.
6. A pattern is worth more than a hundred one-shot tests, because it points at a mechanism.

29.12.2 PLAIN - a picture in your head

1. A doctor cannot find your headache in a ten-minute appointment.
2. So they ask you to keep a diary for two weeks: time, what you ate, how much you slept.
3. The diary finds what the appointment could not, because the fault is a pattern in time, not a state in the room.
4. Network logging is that diary, written automatically once a minute.

Where this comparison breaks: a diary records your own experience, which is continuous. A one-minute network probe samples, so a fault lasting twenty seconds can hide completely between two samples. Sample faster than the fault.

29.12.3 PLAIN - a worked example

1. A logging script. Save it as `netlog.sh`, then run `chmod +x netlog.sh` and `./netlog.sh`.

```
#!/bin/bash
TARGET=github.com
IP=20.207.73.82
LOG=$HOME/netlog.csv
echo "time,dns_ms,tcp,http,rtt_ms" >> "$LOG"
while true; do
  T=$(date -u +%FT%TZ)
  D=$(dig +short "$TARGET" | head -1)
  C=$(nc -z -G 5 "$IP" 443 >/dev/null 2>&1 && echo ok || echo fail)
  H=$(curl -sS -o /dev/null --max-time 10 \
      -w '%{http_code}' "https://$TARGET" 2>/dev/null)
  R=$(ping -c 1 -W 1000 1.1.1.1 2>/dev/null \
      | awk -F'time=' '/time=/{print $2+0}')
  echo "$T,$D,$C,${H:-none},${R:-loss}" >> "$LOG"
  sleep 60
done
```

2. Line by line. `#!/bin/bash` tells the system which interpreter runs the file. `TARGET`, `IP` and `LOG` hold the values you may want to change.
3. `echo "time,..." >> "$LOG"` writes a header row and appends rather than overwriting, because `>>` adds to the end.
4. `while true; do ... done` repeats forever until you press control and C.
5. `T=$(date -u +%FT%TZ)` records the moment in coordinated universal time in a sortable format such as `2026-08-13T09:40:12Z`.
6. `D=$(dig +short "$TARGET" | head -1)` records what the name resolved to, keeping only the first answer, so you can see if the address changed.
7. The `nc -z -G 5` line tests the connection with a five-second limit and records `ok` or `fail`. Sending output to `/dev/null` throws away the chatter and keeps only the success or failure.
8. The `curl` line records the HTTP status code, or nothing if it failed, and `${H:-none}` substitutes the word `none` when the value is empty.
9. The `ping` line takes one measurement to a known-good address and `awk` extracts the number after `time=`, giving you a baseline. `${R:-loss}` writes `loss` when there was no reply.
10. `sleep 60` waits a minute. Lower it to 10 for a fast-moving fault, but be aware that frequent probes can look like scanning to some networks.
11. The result is a comma-separated file you can open in a spreadsheet, sort by time, and plot.

29.12.4 PLAIN - what is really happening inside

1. Why a log beats a test. A test answers "is it broken now". A log answers "when is it broken, and what else is true then".
2. Look for three patterns in the file.
3. Time of day. Failures clustered in the evening point at congestion, since that is when a shared line is busiest.
4. Correlation with your own activity. Failures only during large downloads point at your router's connection table filling up.
5. Uptime correlation. Failures starting a fixed time after every router restart point at a memory or table exhaustion problem in the router.
6. Also watch the resolved address column. If the address changes and the failures follow one particular address, the fault is specific to one front end at the destination, which is a completely different report to write.
7. Continuous path monitoring is the companion tool. `mtr` sends probes repeatedly to every hop and shows loss per hop over time, which catches a flapping link that a single traceroute misses.

29.12.5 TECHNICAL - the engineer's version

1. Continuous path measurement:

```
sudo mtr -n --report --report-cycles 100 20.207.73.82
sudo mtr -n --tcp --port 443 --report-cycles 100 github.com
```

2. `--report` runs a fixed number of cycles and prints a summary table. `--report-cycles 100` sets that number. `--tcp --port 443` probes with TCP SYN packets to the port you actually care about, which is far more representative than ICMP when a filter is port-specific.
3. Read `mtr` loss columns from the bottom up. Loss at an intermediate hop that does not persist to the final hop is control-plane rate limiting, not path loss. Only loss that continues to the destination is real.
4. `mtr` was written by Matt Kimball in 1997 as Matt's Traceroute and has been maintained by Roger Wolff since 1998 under the name My Traceroute.
5. Sampling theory applies. To characterize a fault of duration D with confidence, sample at an interval well below D. A 60-second interval will miss most 20-second events and will under-report their frequency.
6. Prefer recording raw results rather than pass or fail. Store connect time in milliseconds, not a boolean, because degradation precedes failure and only the numeric series shows the trend.
7. To run the logger unattended on macOS, use a `launchd` property list in `~/Library/LaunchAgents` rather than `cron`, because macOS may not run `cron` reliably under modern power management. On Linux, a `systemd` timer or a `cron` entry is the normal choice.
8. Correlate with local link quality on macOS using `sudo wdutil info` for Wi-Fi signal and noise figures. Apple deprecated the older `airport` command-line tool during macOS 14 and it stopped working in a 14.x update in 2024; treat that detail as version-dependent.
9. Apple added `networkQuality` in macOS 12 Monterey in 2021, which measures throughput and responsiveness under load in round trips per minute. Running it during a suspected congestion window is a useful second signal.

29.12.6 WORDS - remember these

1. Intermittent fault — comes and goes — a failure whose probability of observation depends on when you sample.
2. Sampling interval — how often you measure — the gap between probes, which must be shorter than the shortest fault you hope to catch.
3. Flapping — a link going up and down repeatedly — instability that a single traceroute cannot detect but a continuous probe will.

4. Correlation — two things happening together — evidence of a shared cause, not proof of one causing the other.
5. Comma-separated values — a plain text table — one record per line with fields separated by commas, openable by any spreadsheet.
6. Launchd — the macOS way to run things on a schedule — Apple's service manager, configured with property list files, replacing `cron` in practice.

29.13 What to do next in this specific situation

29.13.1 PLAIN - in simple words

1. The reader's fault has a shape: naming works, the connection to one destination on port 443 dies in silence, and the same destination answers from another network.
2. Four hypotheses survive that shape, and no test done so far separates them.
3. So the next steps are chosen for one reason only: each one splits those four hypotheses further.
4. The order below is by cost. Cheap and informative first.
5. Do not skip to contacting the provider. Do it after step 6, with evidence, or you will be sent round the reboot loop.

29.13.2 PLAIN - a picture in your head

1. You know the parcel gets as far as one particular sorting office and then vanishes.
2. Next you would send a second parcel to a different address served by the same office, and a third to the same address by a different route.
3. If the second arrives and the third does not, the office is fine and the address is the problem. If both fail, the office is the problem.
4. That is exactly what the steps below do, with addresses and ports.

Where this comparison breaks: you can watch a parcel leave. Here you must capture your own outgoing packets to know they left at all, which is why a packet capture appears in the list.

29.13.3 PLAIN - a worked example

1. The ordered list.
2. Step 1. Confirm the failure is still happening and is not a memory. Run `curl -v --max-time 20 https://github.com` and keep the output.
3. Step 2. Test the same destination on a different port. Try 22 and 80. If one port opens and 443 does not, the destination is reachable and the fault is port-specific, which is a very different report.
4. Step 3. Test another Microsoft-hosted address, so that most of the path is the same but the machine at the end is not. If that works, the fault is specific to the one address rather than to the whole path.
5. Step 4. Run a TCP traceroute to port 443, which follows the same kind of packet as your real connection instead of the different kind ordinary traceroute uses.
6. Step 5. Capture packets while trying, to prove your machine really sent the connection request and really received nothing.
7. Step 6. If your router allows it, run a ping and a trace from the router itself. That removes your Mac and your Wi-Fi from the experiment entirely.
8. Step 7. Log for an hour with the script from the previous section, so you know whether it is constant or comes and goes.
9. Step 8. Contact the provider with the template from earlier in this chapter, and ask the specific questions listed below.

29.13.4 PLAIN - what is really happening inside

1. Why step 2 matters. A filter usually targets a port. If port 22 opens to the same host and 443 does not, you have proved the route works and a policy is selecting on port.
2. Why step 3 matters. It separates "this path is broken" from "this one machine is not accepting me". Both look identical when you only test one address.
3. Why step 4 matters. Ordinary traceroute on a Mac sends a different kind of packet to unusual port numbers. Some networks treat that differently from real web traffic, which is why a trace can die where your connection would have succeeded, or the reverse.
4. Why step 5 matters. Until you have seen the outgoing packets, "nothing happened" also covers "my own machine never sent anything", which a local firewall or a tunnel interface could cause.
5. Why step 6 matters. If the router fails the same way, every question about your Mac is closed, and the provider cannot blame your device.
6. Why step 7 matters. Providers respond very differently to "it failed once" and "it failed for 47 of 60 consecutive minute-by-minute probes".

29.13.5 TECHNICAL - the engineer's version

1. The commands, in order.

```
# 2. different ports, same host
nc -vz -G 5 github.com 22
nc -vz -G 5 github.com 80
nc -vz -G 5 github.com 443

# 3. different address, same network path
OTHER=$(dig +short www.microsoft.com | tail -1)
nc -vz -G 5 "$OTHER" 443
curl -sSI --max-time 10 https://www.microsoft.com

# 4. TCP traceroute to the port that matters
sudo traceroute -n -P TCP -p 443 20.207.73.82
sudo nmap -Pn -p 443 --traceroute --reason 20.207.73.82
sudo mtr -n --tcp --port 443 --report-cycles 50 20.207.73.82

# 5. prove what left the machine
sudo tcpdump -i en0 -n host 20.207.73.82 and port 443
```

2. On the TCP traceroute: macOS `traceroute -P TCP` exists but its handling of the final hop is unreliable, because a SYN-ACK or RST from the destination is not always recognized as arrival. Prefer `nmap --traceroute` or `mtr --tcp` and treat the macOS built-in as a second opinion.
3. Expected capture signature for a silent drop: repeated outgoing SYN packets to `20.207.73.82:443` with increasing gaps of roughly 1, 2, 4, 8 and 16 seconds under the retransmission rules of RFC 6298, and zero inbound packets from that address. That is the artefact to attach to a ticket.
4. Compare against a working target in the same capture window by adding a second `curl` to a host that works; a capture containing both a successful and a failed handshake side by side is very hard to dismiss.
5. From the router, if it offers a diagnostics page or a shell, run its own ping and traceroute to `20.207.73.82`. Many consumer routers expose exactly these two tools. A failure there removes the Mac, the `utun` interfaces and the local firewall from consideration in one step.
6. Questions to put to the provider, in this wording:

Ask	Why
Route to 20.207.73.82?	Checks forward path
Return route to my prefix?	Checks the invisible half

Ask	Why
Any filter on TCP 443?	Checks policy
Known issue with AS8075?	Checks peering
Can you test from your core?	Removes your line

7. Also ask whether your line is behind carrier-grade network address translation and, if so, which public address you were mapped to at the timestamp you supply. Without that mapping, the far end cannot look you up at all.
8. If the provider confirms nothing and the fault persists, the remaining practical mitigations are a different resolver that returns a different front-end address, an IPv6 path if one can be obtained, or a tunnel that moves your egress to another network. Note that these are workarounds that avoid the fault; none of them fixes it or identifies it.

29.13.6 WORDS - remember these

1. TCP traceroute — a trace that uses real connection packets — probes carrying TCP SYN to a chosen port, so filters that select on port are visible.
2. Egress — where your traffic leaves for the internet — the point and address from which your packets enter the wider network.
3. Capture signature — the recognizable shape in a packet capture — here, retransmitted SYNs with no inbound reply.
4. Carrier-grade NAT — many customers sharing one public address — provider-side address translation, which hides which subscriber a session belonged to.
5. Workaround — avoids the problem without fixing it — a change that restores service by routing around the fault, leaving the cause in place.

29.98 Common wrong ideas

1. Wrong: the site did not load, so the site is down. Right: you proved only that you could not reach it. “Down” is a claim about everybody, and one success anywhere refutes it in ten seconds.
2. Wrong: it worked on mobile data, so my Wi-Fi is the problem. Right: switching to mobile data changes about thirteen variables at once. It proves the destination is alive and nothing about which variable mattered.
3. Wrong: ping works, so the service is fine. Right: ping tests one message type to one address. The service uses a different protocol on a specific port and may be down while ping answers.
4. Wrong: ping fails, so the host is down. Right: very many networks drop ICMP on purpose, and routers deprioritize it. Confirm with a TCP test before concluding anything.
5. Wrong: the stars at the end of my traceroute show where the fault is. Right: silence at the end of a trace is normal. Routers commonly rate-limit or suppress the replies traceroute depends on.
6. Wrong: loss shown at hop 6 means the path is losing packets. Right: read the last hop. Intermediate loss that does not persist to the destination is control-plane rate limiting on that router.
7. Wrong: DNS resolved, so DNS is not involved. Right: DNS resolved to one address out of many possible ones. A different resolver may hand you a different front end that works.
8. Wrong: the status page says all systems operational, so it is my fault. Right: a status page reports service health, not your reachability, and it updates only after a human decides an incident exists.
9. Wrong: a timeout and a connection refused are both just failures. Right: refused means a machine answered and said no. A timeout means nothing answered at all. They point at completely different causes.
10. Wrong: I tested it three times, so I have three pieces of evidence. Right: three identical tests from one vantage point are one observation repeated. Evidence comes from changing a variable, not from repetition.

29.99 Chapter summary in 20 lines

1. Reachability is a property of a pair, a source and a destination, never of a destination alone. That one sentence prevents most wasted hours.
2. "The site is down" is a universal claim; "I cannot reach it" is about you. One success from anywhere refutes the first; your own failure proves the second.
3. The one-line test that separates them is a request to the same destination from a network that shares nothing with yours.
4. The reader ran that test with mobile data and the site loaded instantly, so GitHub was up and the fault was on the reader's own path.
5. That switch changed at least thirteen things: provider, public address, autonomous system, forward path, return path, peering point, resolver, resolved address, address family, middleboxes, path MTU, device stack, time.
6. Only one of them needed to be the cause, so the test had huge power over one question and none at all over which difference mattered.
7. Good diagnosis is a controlled experiment: change one variable, state the hypothesis, and know the eliminating power of the test before running it.
8. The ladder orders the chain: interface, router, public address by number, name resolution, TCP port, full request, path discovery.
9. For every rung, know three things: what a pass proves, what a pass does not prove, and what a failure means. Most mistakes are over-reading a pass.
10. On the reader's ladder, rungs 1 to 4 passed, rung 5 failed with silence, rung 6 never ran, and rung 7 stopped at hop 12 inside Microsoft's Pune site.
11. DNS passing while TCP fails is a very informative pair, because the two used different destinations, protocols and ports; only path, address and port remain as candidates.
12. Silence is the least informative failure mode. No RST, no ICMP, nothing. Consistent with a silent drop on that path, not with the server being down.
13. The trace's `* * *` from hop 13 proves nothing; ICMP time-exceeded is routinely rate-limited, as with the Cisco default of one per 500 ms.
14. Five cross-checks cover every component: other network, other device, other destination, remote shell, and a tunnel. Each eliminates a different family.
15. Ping is ICMP echo, RFC 792 from 1981, written by Mike Muuss in December 1983. It is the most over-trusted tool in networking; use it to measure.
16. TTL in a ping reply lets you count hops; time is a round trip and cannot be halved reliably; loss must be read at the final hop only.
17. On macOS know these flags: `ping -c -i -s -t -W`, `traceroute -n -q -w -m -P`, `dig +short +trace @server`, `nc -vz -G`, `curl -v -I -sS --resolve --max-time -w`. On Linux, `ping -t` means TTL instead, and `-W` is seconds.
18. A fault report needs what you did, what you expected, what happened and what you ruled out, with exact output, a UTC timestamp and a control test.
19. Intermittent faults need logging, not testing: a minute-by-minute log, `mtr` in report mode, and correlation with time of day and load.
20. The reader's remaining steps are TCP traceroute to port 443, a second Microsoft address, a test from the router itself, a packet capture showing retransmitted SYNs with no reply, and a precise question to the provider about the route and the return path towards their own prefix.

30

CHAPTER

KB

How the Internet Was Built

30.0 What this chapter gives you

1. You will be able to tell the story of long-distance messaging from signal fires to fibre, in order, with correct names and years.
2. You will be able to explain packet switching, and say honestly who invented it, when, and why three people arrived at it separately.
3. You will be able to kill the most popular myth in computing: that the ARPANET was built to survive a nuclear war.
4. You will be able to say exactly what happened on 29 October 1969.
5. You will be able to explain what TCP/IP is, why it was split into two layers, and what happened on 1 January 1983.
6. You will be able to state the difference between the internet and the web clearly enough to correct a newspaper.
7. You will be able to explain the dot-com boom and bust with real dates, real numbers, and an honest lesson rather than a moral.
8. You will be able to describe the physical internet: submarine cables, satellites, exchange points, and where India sits in that picture.
9. You will be able to say who actually runs the internet, and why the answer is uncomfortable and interesting at the same time.
10. You will be able to read a modern network path, like the reader's own traceroute through Mumbai and Pune, and see the whole history in it.

30.1 Before electricity: fire, flags and towers

30.1.1 PLAIN – in simple words

1. For most of human history a message travelled at the speed of a horse.
2. There was one exception, found very early: light travels much faster than a person, and a fire on a hill can be seen from the next hill.
3. A chain of fires can carry one fact across a country in minutes.
4. But a fire says only one thing: the agreed event has happened. It cannot say what, or how many, or when.
5. So the next step was to give the signal parts, so it could spell anything.
6. In France in the 1790s, Claude Chappe built towers with two big movable wooden arms on top. Each position of the arms meant a code.
7. A watcher in the next tower copied the shape, and the shape moved down the line. This was the first working long-distance message network in the world, and it ran for over sixty years.

30.1.2 PLAIN - a picture in your head

1. Imagine a line of people on hilltops, each just able to see the next one through a telescope.
2. Each person holds two long sticks. Each stick can be held in several positions, so two arms give many different shapes.
3. Each shape is a number, and both ends of the line hold the same code book. Number 4,102 might mean "the enemy has retreated".
4. Nobody in the middle knows what the message means. They only copy shapes. That is an idea we still use: relaying without understanding.

Where this comparison breaks: there is no second path. A single sleepy operator stops the whole country, and fog, rain and darkness stop everything. A modern network routes around a failure in milliseconds. This one could not route around anything at all.

30.1.3 PLAIN - a worked example

1. Take the Paris to Lille line, the first real Chappe line, opened in 1794.
2. It ran roughly 230 km with about 15 stations, so about 15 km between towers. Sources differ a little on the distance, so treat it as approximate.
3. One symbol took about nine minutes to reach Lille. A full message took about 32 minutes in good weather.

Method	Paris to Lille	Note
Rider on horseback	About 2 days	Roads, weather
Chappe tower line	About 32 minutes	Daylight only
First symbol only	About 9 minutes	Head of message

4. In 1794 the line told Paris that Condé-sur-l'Escaut had been retaken from the Austrians, less than an hour after it happened.

30.1.4 PLAIN - what is really happening inside

1. Strip away the wood and three ideas are already present.
2. **Encoding.** The message becomes symbols the channel can carry. The channel carries shapes, not sentences.
3. **Relaying.** No station talks to the far end. Distance is built out of many short hops between neighbours.
4. **A shared code book.** Both ends must agree in advance what the symbols mean, or the shapes are noise.
5. All three survive today. Encoding is now modulation, relaying is now routing, and the code book is now a protocol specification.
6. The fatal weaknesses were that it was a chain rather than a mesh, and that it carried one message at a time. Packet switching fixes exactly those two things, one hundred and seventy years later.

30.1.5 TECHNICAL - the engineer's version

1. Chappe's first public demonstration was on 2 March 1791 between Brûlon and Parcé, over 16 km, using synchronized pendulum clocks rather than arms.
2. The French Legislative Assembly backed a Paris to Lille relay line in 1792. The line entered service in 1794.
3. Throughput was about 2 to 3 symbols per minute, roughly 0.4 to 0.6 words per minute of plain text.
4. The Chappe code book used 92 basic symbols taken two at a time, giving 8,464 coded words and phrases. That is compression: one symbol pair stands for a whole phrase.
5. At its peak the French network had 556 stations covering about 4,800 km. It stayed in service until 1855, and one of its last messages reported the fall of Sebastopol. Claude Chappe died on 23 January 1805.

Property	Chappe line	Modern comparison
Raw rate	About 0.5 bit/s	1 Gbit/s home fibre
Hop distance	About 15 km	80 to 100 km per span
Availability	Daylight, clear	99.99 percent target
Topology	Single chain	Meshed, multipath

6. The modern term for the tower operator's job is **regenerative repeater**: a device that decides what the symbol was and transmits a clean copy, rather than amplifying the noise along with the signal.

30.1.6 WORDS - remember these

1. Semaphore — signalling by the position of arms — visual signalling using a defined symbol alphabet on a mechanical indicator.
2. Code book — a shared list of what each signal means — a codec or protocol specification agreed by both endpoints.
3. Regenerative repeater — a station that redraws the signal cleanly — a repeater that decides the symbol and retransmits, avoiding noise build-up.

30.2 The electric telegraph

30.2.1 PLAIN - in simple words

1. Electricity moves along a wire almost instantly and does not care about fog, night or hills.
2. In Britain, William Fothergill Cooke and Charles Wheatstone patented a telegraph in May 1837, using needles that swung to point at letters.
3. In the United States, Samuel Morse patented a recording telegraph in 1837. His assistant Alfred Vail built much of the practical machinery.
4. Morse and Vail also produced a code of short and long pulses, which we call Morse code.
5. On 24 May 1844 Morse sent "WHAT HATH GOD WROUGHT" from the Capitol in Washington to Baltimore, over a line paid for by the United States Congress.
6. Wires then spread fast, because railways needed them and newspapers wanted them. The hard part left was the ocean.
7. The first Atlantic cable worked for about three weeks in 1858 and died. A working cable was completed on 27 July 1866, and from that day a message crossed the Atlantic in minutes instead of ten days.

30.2.2 PLAIN - a picture in your head

1. Think of a very long garden hose full of water, with a tap at one end.
2. Open and close the tap quickly and a pattern of pulses comes out the far end. Agree what the patterns mean and you have a telegraph.
3. Now make the hose two thousand miles long and lay it on the sea floor.
4. Water pushed in at one end no longer arrives as a sharp pulse. It arrives smeared out and weak, because the hose absorbs and delays it.
5. Push the pulses closer together and they blur into each other, and the far end cannot tell them apart. That blurring sets how fast you can go.
6. And if you turn the tap on harder to fight the weakness, you can burst the hose. That is roughly what killed the 1858 cable.

Where this comparison breaks: water pressure is not voltage. A long insulated conductor sitting in seawater behaves like a chain of capacitors and resistors, which is why William Thomson, later Lord Kelvin, had to work out the mathematics before long cables could be made to work at all.

30.2.3 PLAIN – a worked example

1. Morse code is not random. Common letters got short codes.

```
E    .
T    -
A    . -
S    . . .
O    - - -
SOS . . .   - - -   . . .
```

2. A skilled operator ran at roughly 20 to 25 words per minute, so that famous four-word phrase took about 10 to 12 seconds to send.
3. Now compare that with the ocean.

Link	Year	Speed
Land line, good operator	1860s	20 to 25 words/min
First Atlantic cable	1858	About 0.1 words/min
Working Atlantic cable	1866	About 8 words/min

4. Queen Victoria's message to President Buchanan on 16 August 1858 was 98 words and took about 16 hours to send. Even that beat a ship by a week.

30.2.4 PLAIN – what is really happening inside

1. A telegraph key is a switch. Pressing it connects a battery to the wire.
2. At the far end an electromagnet pulls a lever whenever current flows. Morse's design pressed that lever against moving paper, so the message was recorded.
3. On a long undersea cable the current does not switch cleanly. The cable stores charge along its whole length, like a very long capacitor.
4. A sharp pulse arrives as a slow rounded hump, and closely spaced humps overlap. That overlap is called **intersymbol interference**, and it is still the limiting problem in every fast link today, including fibre.
5. The 1858 cable failed because the chief electrician, Wildman Whitehouse, applied very high voltage to force a stronger signal and the insulation broke down.
6. The 1866 cable succeeded through better insulation and manufacture and, crucially, instruments sensitive enough to detect a faint signal. On a long link, improving the receiver beats shouting louder.

30.2.5 TECHNICAL – the engineer's version

1. The Cooke and Wheatstone patent of May 1837 recommended five needles, coding 20 of the 26 letters directly.
2. A four-needle system was installed on the Euston to Camden Town section of the London and Birmingham Railway in 1837 and rejected in favour of pneumatic whistles. Their first commercial success was on the Great Western Railway in 1838, over the 21 km from Paddington to West Drayton.
3. Multiplexing arrived because copper and poles were the expensive part.

Technique	Roughly	What it gives
Duplex (Stearns)	1870s	2 messages, both ways
Quadruplex (Edison)	1874	4 messages, 2 each way
Baudot TDM	1874	Several ops share a wire

4. Émile Baudot's 1874 printing telegraph used **time-division multiplexing**: a rotating distributor gave each operator a slot on the same wire. Typical working speed was 30 words per minute.

5. The Baudot code was a fixed-length 5-bit code, unlike Morse's variable length code. Fixed length is what a machine wants; variable length suits a human ear.
6. Cable dates precisely: the 1858 cable was completed on 5 August 1858, first official message 16 August, failed in September after about three weeks. The 1865 attempt broke after 1,062 nautical miles had been paid out from the SS Great Eastern. The successful cable landed on 27 July 1866, from Foilhommerum in Ireland to Heart's Content in Newfoundland.

30.2.6 WORDS - remember these

1. Morse code — dots and dashes for letters — a variable-length code optimized for letter frequency in English.
2. Multiplexing — sharing one wire between several messages — combining channels onto one medium by time, frequency or code.
3. Intersymbol interference — pulses smearing into each other — spreading of symbol energy into adjacent symbol periods, limiting symbol rate.

30.3 The telephone network

30.3.1 PLAIN - in simple words

1. The telegraph sent letters. The next question was whether a wire could carry a voice.
2. Alexander Graham Bell was granted US patent 174,465 on 7 March 1876, and on 10 March 1876 made the first successful call, saying "Mr. Watson, come here, I want to see you".
3. A voice is not dots and dashes. It is a continuously changing signal, so the wire must carry a smooth copy of the sound.
4. At first every telephone needed a direct wire to every other one. Ten phones need forty-five wires. A hundred need 4,950.
5. The fix was an **exchange**: one wire from each phone to a central room, where someone joins two wires together.
6. Almon Strowger, an undertaker, believed operators were sending his calls to a rival, and built a machine to replace them. His patent was granted in 1891 and the first automatic exchange opened in La Porte, Indiana on 3 November 1892, for about 75 subscribers.
7. From that grew the largest machine humans had built, working in a way that could never have become the internet.

30.3.2 PLAIN - a picture in your head

1. Think of a railway where the whole route is booked for one train.
2. Every section of track from your station to your destination is reserved before you leave, and nobody else may use any part of it.
3. That is **circuit switching**. It is wonderful for a phone call, because once booked the quality never wavers mid-sentence.
4. It is terrible for computers, which send a burst, think for four seconds, then send another burst.

Where this comparison breaks: from the 1960s, long-distance calls were carried digitally over shared links in time slots, so there was no continuous copper path. But the reservation model stayed: a slot was booked for the whole call whether you spoke or not. The reservation, not the copper, is the point.

30.3.3 PLAIN - a worked example

1. Suppose you type at 5 characters per second to a remote machine. At 8 bits per character that is 40 bits per second of real information.
2. The circuit reserved for you is a digital voice channel, 64,000 bits per second.
3. You are using about 0.06 percent of it. The rest is reserved and wasted.
4. Now put 100 such users side by side.

Model	Reserved for 100	Actually used
Circuit switching	6,400,000 bit/s	4,000 bit/s
Packet switching	64,000 bit/s	4,000 bit/s

5. That table is the entire economic argument for the internet.
6. It is also why the telephone companies, who were excellent engineers, resisted the idea. Their business was selling reserved time.

30.3.4 PLAIN - what is really happening inside

1. Lift a handset and you close a loop, so direct current flows from the exchange through your phone and back.
2. A rotary dial breaks that current once per unit of the digit. Dial 5 and it breaks the loop 5 times.
3. A Strowger switch counts the breaks with an electromagnet, stepping the wiper up five levels. The next digit rotates it to one of ten contacts, so two digits select one of a hundred outlets.
4. Nothing in that path stores your voice or looks at it. The network is a wire-joining machine, not a message-handling machine, and that is the deep reason it could not become the internet.

30.3.5 TECHNICAL - the engineer's version

1. Elisha Gray filed a caveat for a liquid-transmitter telephone on 14 February 1876, the day Bell's application was filed. Gray did not contest priority, and the examiner approved Bell's patent on 3 March 1876.
2. The Bell Telephone Company was formed in 1877. By 1886 over 150,000 Americans had telephones.
3. Strowger conceived his switch in 1888 and received US patent 447,918 in

1891. The prototype used a collar box and pins. The two-motion selector gives 10 levels by 10 contacts, so 100 outlets.

4. Digital voice standardized on 8,000 samples per second at 8 bits, giving the 64 kbit/s DS0 channel. That sampling rate follows from the Nyquist rule applied to a 300 to 3,400 Hz voice band.

Level	Region	Rate
DS0	Worldwide	64 kbit/s
T1 (DS1)	North America	1.544 Mbit/s
E1	Europe, India	2.048 Mbit/s
T3 (DS3)	North America	44.736 Mbit/s

5. Signalling System No. 7 moved call setup onto a separate packet network while voice used reserved circuits. Note the detail: SS7 is itself packet based.
6. The Bell System was broken up by consent decree effective 1 January 1984, separating AT&T from the regional operating companies.
7. Stated technically, the mismatch is this: circuit switching gives constant bit rate, constant delay and zero statistical multiplexing gain. Computer traffic is bursty, often with a peak-to-mean ratio above 100 to 1, and gains enormously from statistical multiplexing.

30.3.6 WORDS - remember these

1. Exchange — the building that joins calls — a switching centre terminating subscriber loops and interconnecting them.
2. Circuit switching — booking a path for the whole call — dedicating a fixed channel end to end for a session's duration.
3. DS0 — one digital voice channel — 64 kbit/s, 8,000 samples per second at 8 bits, the base unit of the digital telephone hierarchy.

30.4 Packet switching, invented three times

30.4.1 PLAIN - in simple words

1. Packet switching is the idea everything on the internet rests on.
2. Instead of booking a path and sending a stream, you chop the message into small pieces.
3. Each piece carries the full destination address, like an envelope with a complete postal address on it.
4. Each piece travels on its own. Machines along the way read the address and pass it towards the destination.
5. Different pieces may take different routes, arrive out of order, or be lost. The receiving end reorders them and asks for anything missing.
6. Nobody books anything. The link is shared moment by moment between whoever has a piece ready to send.
7. Three people worked this out separately in the early 1960s, in two countries, for two completely different reasons. When an idea is genuinely ready, it tends to be found more than once.

30.4.2 PLAIN - a picture in your head

1. You want to send a long book to a friend in another city.
2. Circuit switching is hiring a private van that drives non-stop from your door to theirs with the whole book inside.
3. Packet switching is tearing the book into single pages, putting each page in its own addressed envelope with a page number, and posting them all.
4. The envelopes mix with everyone else's post. Each sorting office moves each one closer. Some go one way, some another, and they arrive out of order.
5. Your friend sorts them by page number. If page 74 never turns up they ask for page 74 only, not the whole book.

Where this comparison breaks: the post has no congestion feedback. The internet does. When routers fill up and drop packets, senders slow down. That feedback loop is why your download speed changes second by second, and nothing in the postal picture explains it.

30.4.3 PLAIN - a worked example

1. Take a 10,000-byte file on a network whose maximum packet size is 1,500 bytes, the common Ethernet limit.
2. It splits into 7 packets: six of 1,500 bytes and one of 1,000.
3. Each gets a header. A minimal IPv4 header is 20 bytes and a minimal TCP header another 20, so 40 bytes of addressing and control per packet.

```
File 10,000 bytes
  +-> pkt 1..6  [hdr 40][data 1500]
  +-> pkt 7     [hdr 40][data 1000]
On the wire: 10,000 + 7*40 = 10,280 bytes
Overhead:    280 / 10,280 = 2.7 percent
```

4. Overhead is 2.7 percent. That is the price of the whole idea, and it is small.
5. Now lose packet 4. Circuit switching has no answer, because it assumed the path was reliable.
6. Packet switching resends 1,540 bytes instead of 10,280. A 15 percent cost instead of a 100 percent cost.

30.4.4 PLAIN – what is really happening inside

1. The three inventors reached the same design from different starting points, and it helps to keep them apart.
2. **Paul Baran**, at the RAND Corporation, worked from around 1960 on survivable communications. He proposed a distributed mesh with many alternative paths, messages cut into fixed **message blocks**, and store-and-forward switching with what he called hot potato routing.
3. **Donald Davies**, at the National Physical Laboratory in Britain, started in 1965 from a different problem: interactive computing was painful because telephone lines were slow to set up and wasteful when idle.
4. Davies coined the word **packet**, choosing a short word that would survive translation into other languages.
5. **Leonard Kleinrock**, at MIT and then UCLA, did the mathematics of queues: what happens to delay when messages wait in line at a busy node.
6. The credit question got heated later. The honest position is that historians and primary sources credit Baran and Davies with independently inventing packet switching.
7. Kleinrock's early published work was about message switching and queueing delay, and in 2023 he acknowledged that. Donald Davies wrote that he could find no evidence Kleinrock understood the principles of packet switching from that work, and Alex McKenzie of BBN said the 1964 book contains nothing analysing packetization.
8. That is a live disagreement between people who were there. State it as a disagreement, not as a settled insult in either direction.

30.4.5 TECHNICAL – the engineer's version

1. Baran's work at RAND was first briefed to the US Air Force in summer 1961 as briefing B-265, published as RAND paper P-2626 in 1962 and as the report series RM-3420 in 1964. The full series, "On Distributed Communications", ran to eleven volumes.
2. His three core ideas were a decentralized network with multiple paths between any two points, division of user messages into message blocks, and delivery by store-and-forward switching.
3. Davies conceived his design in 1965, proposed a national commercial data network publicly in 1966, and his colleague Roger Scantlebury presented the work at the Symposium on Operating System Principles in October 1967. The NPL Data Communications Network piloted in early 1969 and entered service in 1970.

Person	Where	Contribution
Paul Baran	RAND, USA	Distributed mesh, blocks
Donald Davies	NPL, UK	The packet, network design
Leonard Kleinrock	MIT, UCLA	Queueing analysis

4. Now the nuclear myth, stated precisely, because it needs care.
5. It is **true** that Baran's funding context was US Air Force interest in communications that could survive attack, and his reports address that.
6. It is **false** that the ARPANET was built for that reason. ARPA director Charles Herzfeld stated plainly that the ARPANET was not started to create a command and control system that would survive a nuclear attack.
7. The ARPANET's real motivation was resource sharing: expensive research computers were scarce, incompatible and scattered.
8. So the correct sentence is that one of the three independent inventions of packet switching had a survivability motive, and the network that first deployed it did not.
9. The model that matters in practice is Little's Law: L equals lambda times W, relating mean number in system, arrival rate and mean time in system. It is why a link driven near 100 percent utilization has unbounded queueing delay.

30.4.6 WORDS - remember these

1. Packet — a small piece of a message with an address on it — a self-contained protocol data unit with header and payload.
2. Store and forward — receive it fully, then send it on — a node buffers the whole unit before retransmitting on the next hop.
3. Little's Law — long queues mean long waits — L equals lambda times W.

30.5 ARPANET

30.5.1 PLAIN - in simple words

1. ARPA was a research funding agency inside the United States Department of Defense.
2. In October 1962 J.C.R. Licklider was put in charge of its computing office. He had already published "Man-Computer Symbiosis" in 1960.
3. On 23 April 1963 he sent a memo addressed to "Members and Affiliates of the Intergalactic Computer Network". The joke name carried a serious idea: these separate machines ought to be able to talk to each other.
4. In 1966 Bob Taylor, then running that office, started the actual project. He was irritated by having three terminals on his desk for three remote computers.
5. He hired Larry Roberts to run it, and the work went out to tender in 1968.
6. A small company in Cambridge, Massachusetts called Bolt Beranek and Newman won the contract in January 1969.
7. They built a box called an Interface Message Processor, which sat between a university's computer and the network. On 29 October 1969 the first message was sent, and the receiving machine crashed after two letters.

30.5.2 PLAIN - a picture in your head

1. Imagine four universities, each with one expensive and completely different computer, with different operating systems and tape formats.
2. So instead you put a small identical box next to each computer.
3. Every box is the same model from the same company running the same software, and the boxes talk to each other over leased telephone lines.
4. Each box talks to its own local computer in that computer's own language.
5. The universities never had to agree with each other. Each only had to agree with the box in its own machine room.

Where this comparison breaks: the box was not merely a translator. It did the routing, buffering, error checking and retransmission. It was the first router, and it did the hard part so the host computers could stay ignorant of it.

30.5.3 PLAIN - a worked example

1. On 29 October 1969, at about 10:30 pm Pacific time, Charley Kline at UCLA tried to log in to the SRI machine over the new network.
2. To log in he had to type LOGIN, and the remote system echoed each character.

```
UCLA (SDS Sigma 7)          SRI (SDS 940)
    |-- 'L' ------------------>  |
    |  <---------------- echo 'L'|
    |-- 'O' ------------------>  |
    |  <---------------- echo 'O'|
    |-- 'G' ------------------>  |
    |                       [crash]
```

3. So the first message ever sent between two computers on the ARPANET was "LO". The crash was in SRI's host software, not in the network, and a full login worked about an hour later.
4. The four original nodes, installed through the last months of 1969, were these.

Node	Host computer
UCLA	SDS Sigma 7
SRI (Stanford Research)	SDS 940
UC Santa Barbara	IBM 360/75
University of Utah	DEC PDP-10

30.5.4 PLAIN - what is really happening inside

1. The Interface Message Processor, always called the IMP, was a ruggedized Honeywell DDP-516 minicomputer with 24 kilobytes of magnetic core memory.
2. Each IMP connected to its local host on one side and to two or more other IMPs on the other, over leased telephone lines.
3. Each IMP kept a table saying which neighbour was the best next hop for each destination, updating it as conditions changed. That is exactly what a router does today; the word did not exist yet.
4. Above the IMP layer, host computers spoke the Network Control Program, or NCP, which assumed the network underneath was reliable.
5. That assumption is the key difference from TCP/IP and is what eventually had to be thrown away. Note the split, though: identical switches doing the networking, wildly different hosts doing the applications. That is still the shape of the internet.

30.5.5 TECHNICAL - the engineer's version

1. The honest version of the dates: Licklider arrived at the Information Processing Techniques Office in October 1962, and the famous Intergalactic Computer Network memo is dated 23 April 1963. Use 1962 for his arrival and 1963 for the document.
2. The request for quotation went to 140 potential bidders in 1968 and 12 bid. BBN won, and the contract was awarded in January 1969.
3. The host-to-IMP interface was specified in BBN Report 1822, which is why engineers still call it 1822. Lines between IMPs ran at 50 kbit/s.
4. The public demonstration was at the International Conference on Computer Communications in Washington in October 1972, organized largely by Robert Kahn, with about 40 machines connected live in a hotel.
5. That demonstration matters more than it sounds. Until then, telephone engineers widely considered packet switching doubtful. Afterwards it was considered obvious.

Date	Rough scale
Dec 1969	4 nodes
1971	About 15 nodes
1973	First links outside USA
1981	213 hosts

6. In 1973 the ARPANET gained links to Norway, via NORSAR, and to University College London, making it international.
7. The tools that echo this era are still on your machine. `ping` measures round-trip time and `traceroute` shows the hop sequence, because a hop-by-hop store-and-forward network needs hop-level diagnostics.

30.5.6 WORDS - remember these

1. ARPANET — the first packet switched network of this kind — the ARPA-funded research network, operational 1969 to 1990.
2. IMP — the box that did the networking — Interface Message Processor, a Honeywell DDP-516 acting as the first packet switch, specified in BBN Report 1822.
3. NCP — the early host-to-host rules — Network Control Program, the ARPANET host protocol before TCP/IP, which assumed a reliable subnet.

30.6 Email and the applications that came first

30.6.1 PLAIN - in simple words

1. The ARPANET was funded to let people share expensive computers. What people actually did with it, almost immediately, was send each other messages.
2. Machines already had local mail: you could leave a note for another user of the same computer.
3. In 1971 Ray Tomlinson at BBN joined a local mail program to a file transfer program, so a note could be left on a different computer.
4. He needed a way to write "this user, on that machine", and picked the @ sign because it was not used in user names and read naturally as "at".
5. Email spread on its own, with nobody planning or funding it, and within a couple of years it was the bulk of the traffic.
6. That is the pattern to remember: the important application is usually not the one in the funding proposal.

30.6.2 PLAIN - a picture in your head

1. Think of a research building where every office has a pigeonhole for internal notes. That works, but only inside the building.
2. Now someone realizes the courier who carries parcels between buildings could also carry notes.
3. So they write a note, put it in a parcel, and mark it for the pigeonhole of Dr Rao, in the Chennai building.

Where this comparison breaks: unlike a parcel, an email is copied rather than moved, so the sender still has it. That single difference is why email became a permanent archive, and why mailing lists were possible at all.

30.6.3 PLAIN - a worked example

1. An email address has exactly two parts, split by the @ sign. In `tomlinson@bbn-tenexa`, the local part is `tomlinson` and the host part is `bbn-tenexa`.
2. Here is a modern message as it actually travels.

```
S: 220 mail.example.net ESMTP ready
C: EHLO client.example.org
S: 250 STARTTLS
C: MAIL FROM:<student@example.org>
S: 250 2.1.0 Ok
C: RCPT TO:<teacher@example.net>
S: 250 2.1.5 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: Subject: chapter 30
C: (blank line, then the body)
C: .
S: 250 2.0.0 Ok: queued as 4A2F1C
C: QUIT
```

3. Notice the envelope and the content are separate. `MAIL FROM` and `RCPT TO` are the envelope; the `Subject` line inside `DATA` is content.
4. Mail is delivered to the envelope address, not the `To:` header. That is why blind copies work at all.
5. That design is from RFC 821 in 1982, and the modern version is RFC 5321 from 2008. The shape has not changed.

30.6.4 PLAIN - what is really happening inside

1. Three applications defined the first decade.
2. **Telnet** gave you a terminal session on a remote machine. **FTP**, the File Transfer Protocol, moved files; its first specification is RFC 114 from 1971. **Mail** began as a bolt-on to file transfer and then got its own rules.
3. MsgGroup started in 1975. Another early list, SF-Lovers, was about science fiction, and administrators argued about whether a defence-funded network should carry it. That argument about acceptable use returns in the 1990s and matters enormously.
4. In 1979 and 1980, Tom Truscott and Jim Ellis at Duke University, with Steve Bellovin, built something different: **Usenet**.
5. Usenet was not on the ARPANET. It ran over ordinary dial-up calls between Unix machines using UUCP, Unix-to-Unix Copy. Each machine phoned its neighbours and swapped new articles, so over days an article reached every participating machine in the world.
6. That is store-and-forward at the application level, with no central server, no live connection and no permission needed from anybody. It is where the words FAQ, flame war and spam entered common use.

30.6.5 TECHNICAL - the engineer's version

1. Tomlinson's 1971 work combined SNDMSG, a local message program, with CPYNET, an experimental file copy program, on TENEX systems at BBN. The exact date is not recorded, only the year.

Function	Then	Now
Remote login	Telnet, RFC 854	SSH, RFC 4251
File transfer	FTP, RFC 959	HTTPS, SFTP
Mail transport	RFC 821 (1982)	RFC 5321 (2008)
Mail format	RFC 822 (1982)	RFC 5322 (2008)

2. Usenet was publicly established in 1980 at Duke University and the University of North Carolina at Chapel Hill. The first group was net.general.
3. The Usenet transport protocol over TCP is NNTP, RFC 977 from 1986, updated by RFC 3977 in 2006.
4. Email's structural weakness was baked in early. SMTP has no authentication of the sender, and everything added since, SPF (RFC 7208), DKIM (RFC 6376) and DMARC (RFC 7489), is a patch over that original openness.
5. You can still watch the protocol directly with `openssl s_client -starttls smtp -connect mail.example.net:587`.

30.6.6 WORDS - remember these

1. Email — a note left in a mailbox on another computer — a store-and-forward messaging system using SMTP for transport and RFC 5322 for format.
2. Envelope — the delivery instructions, separate from the letter — the SMTP MAIL FROM and RCPT TO commands, distinct from message headers.
3. Usenet — a worldwide notice board copied machine to machine — a distributed discussion system using flood-fill propagation over UUCP or NNTP.

30.7 TCP/IP and the flag day

30.7.1 PLAIN - in simple words

1. By the early 1970s there was not one network but several, and they were all different.
2. The ARPANET ran over leased telephone lines, PRNET was a radio network in the San Francisco area, and SATNET went over satellite to Europe.
3. The question was how to join networks that disagree about everything.
4. Bob Kahn posed the problem and Vint Cerf worked on it with him from spring
1973. In May 1974 they published "A Protocol for Packet Network Intercommunication" in IEEE Transactions on Communications.
5. Their answer was not to make the networks agree. It was to put a common protocol on top of all of them, with boxes called gateways in between.
6. That is what the word internet means: internetworking, networks of networks. In 1978 the design was split into TCP and IP, and on 1 January 1983 the ARPANET switched to them in a single day.

30.7.2 PLAIN - a picture in your head

1. Imagine shipping goods across a continent where each country has different trucks, roads and load limits.
2. The good solution is the shipping container. Agree only on the box.
3. IP is the shipping container: a standard box with an address, able to travel over any underlying network.
4. TCP is the paperwork saying how many boxes there are, in what order, and what to do if one is lost. Gateways are the border crossings.

Where this comparison breaks: a real container is never opened at the border, but an IP packet may be cut into smaller pieces if the next network cannot carry one that big. That is fragmentation, and it is one of the messier parts of the design. Also, no shipping firm would tolerate quietly losing containers. IP does, on purpose, leaving recovery to TCP.

30.7.3 PLAIN - a worked example

1. Look at what happened on the reader's own machine when it tried to reach `github.com`, which DNS resolved to `20.207.73.82`.

```
 1  192.168.0.1          home router
 2  172.31.0.17          ISP private core
 3  137.97.29.249        ISP public edge
 7  ...del01...msn.net   Microsoft, Delhi
 8  ...bom01...msn.net   Microsoft, Mumbai
10  ...pnq21...msn.net   Microsoft, Pune
13+ * * *                no reply
```

2. Hops 1 to 6 are inside the reader's ISP. Hops 7 onwards are inside Microsoft's own backbone. Two completely separate organizations.
3. Nothing there required them to run the same equipment, software or internal addressing. They only had to agree on the IP packet format and on how to exchange routes.
4. Then look at the failure. The connection to port 443 timed out after 15 seconds with no reply of any kind: no reset, no error, silence.
5. IP did its job by promising nothing. TCP did its job by retrying and then giving up. Correct layering does not mean success. It means each layer fails in a well-defined way.

30.7.4 PLAIN - what is really happening inside

1. The original 1974 design was one protocol doing both jobs, addressing and reliability.
2. That was wrong for one important reason: not every application wants reliability. Digitized voice would rather lose a packet than wait for it.
3. So in 1978 the design was split. **IP** does addressing and forwarding and promises nothing. **TCP** sits on top and adds order, retransmission and flow control.
4. IP is deliberately **best effort**: it will try, it may fail, and it will not tell you.
5. This is the hourglass or thin waist model: many technologies below, many applications above, one narrow protocol in the middle. That narrow waist is why the internet survived every change underneath it, from 50 kbit/s leased lines to 400 Gbit/s optics, without rewriting the applications.
6. The companion idea is the **end-to-end principle**: put function in the endpoints, not the network, unless there is a strong reason. That is why routers are simple and your laptop is complicated. The telephone network did the opposite, and could not change.

30.7.5 TECHNICAL - the engineer's version

1. Cerf and Kahn, "A Protocol for Packet Network Intercommunication", IEEE Transactions on Communications, volume 22 issue 5, pages 637 to 648, May 1974.
2. RFC 675, December 1974, by Cerf, Dalal and Sunshine, was the first detailed specification and is where "internet" appears as a contraction of internetwork.
3. Version 4, in 1978, split the Transmission Control Program into IP, connectionless, and TCP, connection-oriented. Jonathan Postel drove the split. That version number is why we have IPv4 and never deployed an IPv1, 2 or 3.
4. The core specifications settled as RFC 791 (IP) and RFC 793 (TCP), both September 1981, plus RFC 768 (UDP, 1980) and RFC 792 (ICMP, 1981).
5. A **flag day** is a cutover with no compatibility period: at a set instant everyone stops the old thing and starts the new one, with no fallback.
6. The NCP to TCP/IP flag day was 1 January 1983. It was announced years ahead, tested in stages during 1982, and enforced by turning NCP off. Sites that were not ready lost service.
7. It worked because the network had a few hundred hosts, one funding agency could compel every site, and every administrator was reachable by name.

Property	Jan 1983	Today
Hosts affected	A few hundred	Billions
Single authority	Yes, ARPA	No
Rollback plan	Turn NCP back on	None possible
Migration style	Flag day	Dual stack, decades

8. IPv6 proves the point. It was specified in RFC 1883 in 1995, updated as RFC 2460 in 1998 and RFC 8200 in 2017, and adoption is still partial in 2026, with Google's measurement of IPv6-capable users in the range of roughly 40 to 50 percent in recent years.
9. The reader's machine reported `IPv6: (none)`, an ordinary result on many Indian home connections, which illustrates the point exactly.
10. Instead of flag days we use **dual stack**, running both protocols at once, with transition mechanisms such as NAT64 and 464XLAT. The cost is running two internets in parallel for thirty years.

30.7.6 WORDS - remember these

1. Internetworking — joining different networks into one — providing a uniform service across heterogeneous networks via a common protocol.
2. IP — the addressed envelope — the connectionless best-effort internet layer protocol, RFC 791 for version 4 and RFC 8200 for version 6.

3. TCP — the part that makes sure everything arrives in order — a connection-oriented reliable byte stream with congestion control.
4. Flag day — everyone switches at the same instant — a cutover with no backward compatibility period.
5. End-to-end principle — keep the clever parts at the edges — place function in end hosts unless it cannot correctly be implemented there.

30.8 Other networks and the standards war

30.8.1 PLAIN - in simple words

1. It is easy to think the internet was the only game in town. It was not.
2. Through the 1980s there were many networks, most of them bigger and better funded.
3. Telephone companies offered **X.25**, a public data network standard first published in 1976.
4. France gave away terminals for a service called **Minitel** and had millions of users while most of the world had none.
5. Universities that could not join the ARPANET built **BITNET** in 1981, and hobbyists built **FidoNet** from 1984, linking home bulletin boards by nightly phone calls.
6. CompuServe and America Online sold access to their own private worlds, with their own content and their own rules.
7. And an enormous international standards effort, **OSI**, was supposed to produce the one true networking suite. Governments mandated it, textbooks taught it, and TCP/IP beat it completely.

30.8.2 PLAIN - a picture in your head

1. Imagine two teams designing a bicycle for the whole world.
2. Team B builds a rough bicycle, rides it, fixes the problems, and gives away both the plans and a working example for free.
3. Ten years later everyone rides Team B's bicycle, and Team A's document sits on a shelf being technically excellent.

Where this comparison breaks: OSI was not a failure of intelligence. Its vocabulary is still how we teach networking, and the seven-layer model is genuinely useful even though the seven-layer protocols lost. The failure was of process and timing, not of ability.

30.8.3 PLAIN - a worked example

1. Compare the rivals of the 1980s side by side.

Network	Started	Fate
X.25 public networks	1976	Replaced, niche use
Minitel (France)	1982	Closed 30 June 2012
BITNET	1981	Support ended 1996
FidoNet	1984	Tiny, still alive

2. Minitel is the most instructive. France Telecom gave terminals away free from 1984, and by early 1986 there were 1.4 million connected.
3. By the end of 1999 there were close to 9 million terminals used by around 25 million people, with revenue of 620 million francs in 1985 and about 832 million euros in 1998.
4. The reason matters: Minitel was closed, with one operator deciding what services could exist. The web let anybody publish without permission.
5. FidoNet is the opposite lesson. With no money at all it peaked at around 39,000 systems in the mid 1990s and roughly 4 million users by 1993, using hierarchical addresses like `1:250/250.10`.

30.8.4 PLAIN - what is really happening inside

1. To understand why TCP/IP won, separate four advantages that people usually collapse into one.
2. **Running code.** TCP/IP existed and worked while OSI was still being written. In particular the 4.2BSD release of Berkeley Unix in 1983 shipped a usable TCP/IP stack with source, on the machines universities were already buying.
3. **No licence fees and no owner.** RFCs were free to read and free to implement. ISO standards had to be purchased, and still do.
4. **Government backing of the useful kind.** The US Department of Defense required TCP/IP on its networks, which created a large market immediately. OSI mandates promised a market later.
5. **Simplicity.** IP has one job. OSI's network layer had modes, options and variants, so two implementations could both be correct and still fail to interoperate.
6. The honest counter-argument, still made by experts: OSI had better answers for network management, addressing scale and formal description, and some of the pain of the 1990s came from TCP/IP's weaker versions of those. The better process beat the better committee, and we inherited both the benefits and the gaps.

30.8.5 TECHNICAL - the engineer's version

1. X.25 was finalized by CCITT in 1976 as the Orange Book, revised in 1980, 1984, 1988, 1993 and 1996. It provided virtual circuits, typically at 2,400 bit/s to 64 kbit/s, rarely above.
2. BITNET was founded in 1981 by Ira Fuchs of CUNY and Greydon Freeman of Yale, first linking those two. It used IBM's RSCS and NJE protocols, peaked around 1991 at almost 500 organizations and about 3,000 nodes, and lost support in 1996. LISTSERV came out of that world.
3. FidoNet was created by Tom Jennings in San Francisco in early 1984, first released in June 1984. Zone Mail Hour was a reserved period, commonly around 04:00, when nodes exchanged mail and human callers were locked out.
4. OSI: ISO began the work in 1977; the reference model was published as ISO 7498 in 1984, after a merged CCITT and ISO text was agreed in May 1983.

OSI layer	Number	TCP/IP reality
Application	7	HTTP, SMTP, DNS
Presentation	6	Mostly in the app
Session	5	Mostly in the app
Transport	4	TCP, UDP
Network	3	IP
Data link	2	Ethernet, Wi-Fi, PPP
Physical	1	Copper, fibre, radio

5. The US Government OSI Profile, GOSIP, became a federal procurement requirement in 1990 and was effectively abandoned in 1994 and 1995 when agencies were allowed to buy TCP/IP instead.
6. ASN.1, an OSI-era notation, is everywhere in security: the X.509 certificates your browser checks on every HTTPS connection are defined in ASN.1 and encoded in DER.
7. So the accurate summary is not that OSI failed. It is that the OSI protocol stack lost and several OSI ideas won quietly.

30.8.6 WORDS - remember these

1. X.25 — the phone companies' data network — a CCITT virtual-circuit packet network standard from 1976.
2. Walled garden — a service where one company decides what exists inside — a closed platform with controlled content and gated third-party access.

3. OSI — the official international networking suite — the ISO and CCITT protocol family and its seven-layer reference model, ISO 7498.

30.9 From research network to public internet

30.9.1 PLAIN – in simple words

1. Until the late 1980s you could not simply buy internet access. There was nothing to buy.
2. You got on the internet by being at a university or research lab with a connection paid for by a government agency.
3. In 1986 the US National Science Foundation built **NSFNET** to link supercomputer centres and then regional university networks.
4. It came with an **acceptable use policy** saying the network was for research and education. Commercial traffic was not allowed.
5. So a national network existed and businesses were forbidden to use it. They built their own connections and their own exchange points instead.
6. Between 1991 and 1993 the restrictions were relaxed and then removed, and commercial providers appeared quickly.
7. On 30 April 1995 the NSFNET backbone was switched off. That date is a good candidate for the day the internet became a public utility rather than a research project.

30.9.2 PLAIN – a picture in your head

1. Imagine a country with excellent motorways built only for university buses. Lorries are banned.
2. Eventually the private roads are better than the public one, the government closes its motorway, and everyone drives on the private roads.
3. The interchanges remain, and they are now the most important places on the map.

Where this comparison breaks: unlike a road interchange, two networks meeting at an exchange point may simply refuse to exchange traffic with each other. Peering is voluntary, and refusals happen, sometimes publicly and expensively.

30.9.3 PLAIN – a worked example

1. Follow the NSFNET backbone as demand grew.

Year	Backbone	Speed
1986	6 sites, leased	56 kbit/s
July 1988	13 nodes	1.544 Mbit/s
Nov 1991	Upgraded	45 Mbit/s
Apr 1995	Decommissioned	Handed to industry

2. Put a modern number beside it. DE-CIX Frankfurt, one exchange point, recorded a peak of 18.73 Tbit/s in December 2025.
3. That is roughly 416,000 times the entire 1991 national backbone, in one building, in one city.
4. That is why the acceptable use policy felt reasonable at the time. The capacity genuinely could not have absorbed commercial traffic.

30.9.4 PLAIN – what is really happening inside

1. The acceptable use policy was not a law. It was a condition attached to money, which is often stronger.
2. Commercial networks that wanted to exchange traffic without touching NSFNET formed the Commercial Internet eXchange, CIX, in 1991.

3. That is the first clear example of **settlement-free peering**, still how most large networks interconnect.
4. In 1991 the NSF removed access restrictions, and on 23 October 1992 US legislation authorized the NSF to support networks used substantially for purposes beyond research and education.
5. For the handover, NSF funded four **Network Access Points**, in New York, Washington DC, Chicago and San Jose. They were transitional and were replaced by hundreds of independent exchange points worldwide.
6. The structural change is the important part. Before 1995 there was a backbone. After 1995 there is no backbone, only a mesh of networks that agree to carry each other's traffic.

30.9.5 TECHNICAL - the engineer's version

1. NSFNET began in 1986 with six backbone sites on leased 56 kbit/s links. The T-1 backbone, 1.544 Mbit/s across 13 nodes, was in place by July 1988, operated by Merit Network with IBM and MCI. The T-3 backbone at 44.736 Mbit/s was deployed in November 1991.
2. The NSFNET Backbone Service transition completed and the fibre backbone was decommissioned on 30 April 1995.
3. What replaced it is described by business relationship, not technology.

Relationship	Money flows	Routes given
Transit	Customer pays	Full table
Settlement-free peer	Nobody pays	Own customers only
Paid peering	One side pays	Own customers only

4. Those relationships are enforced in BGP, the Border Gateway Protocol, currently RFC 4271. BGP decided the path the reader's packets took towards Microsoft.
5. An autonomous system number, an ASN, identifies each independently routed network. Roughly 75,000 to 80,000 ASNs are visible in the global routing table in recent years, and the IPv4 table passed 1,000,000 routes in 2025.
6. An exchange point is physically a large Ethernet switching fabric in a neutral building. Members take a port, commonly 10, 100 or 400 Gbit/s, and set up BGP sessions with whichever other members agree.
7. DE-CIX Frankfurt had more than 4,300 connected networks by the end of 2025.
8. India's National Internet Exchange, NIXI, was registered on 19 June 2003 specifically so Indian-to-Indian traffic would stop being hauled via Singapore, London or the United States. It has reported peak traffic of about 393 Gbit/s inbound.
9. DE-CIX opened its first Indian exchange in Mumbai in 2018 and now operates in several Indian cities including Delhi, Chennai, Bengaluru, Hyderabad and Kolkata.

30.9.6 WORDS - remember these

1. NSFNET — the American research backbone of the late 1980s — the NSF-funded TCP/IP backbone, 1986 to 30 April 1995.
2. Acceptable use policy — the rule saying what the network may be used for — a funding condition restricting traffic to research and education.
3. Peering — two networks swapping traffic directly — a BGP interconnection, usually settlement-free and carrying customer routes only.
4. Internet exchange point — a building where many networks meet — a shared layer 2 fabric enabling many-to-many BGP peering.
5. Autonomous system — one network under one routing policy — an independently administered routing domain identified by an ASN.

30.10 The World Wide Web

30.10.1 PLAIN - in simple words

1. By 1989 the internet existed and worked, but there was no web.
2. To read a document on another machine you had to know it existed, know which machine held it, log in, find the directory and fetch the file.
3. Tim Berners-Lee was a software engineer at CERN, the particle physics laboratory near Geneva, where thousands of scientists came and went.
4. In March 1989 he wrote a proposal called "Information Management: A Proposal", suggesting a linked information system. He wrote a second in May 1990, and a formal one with Robert Cailliau on 12 November 1990.
5. By the end of 1990 he had written a server and a browser, both running on a NeXT computer, and the first website was live at `info.cern.ch`.
6. On 6 August 1991 he posted about the project publicly, and the software went out to the whole internet on 23 August 1991.
7. On 30 April 1993 CERN put the web software into the public domain, free for anyone to use with no fee and no permission. That may be the most valuable giveaway in the history of technology.

30.10.2 PLAIN - a picture in your head

1. Think of a library with millions of books and no catalogue.
2. Now each book gains two things. First, a printed address on its spine that identifies it uniquely worldwide.
3. Second, wherever a book mentions another book, that mention becomes a button. Press it and the other book appears in your hands.

Where this comparison breaks: real links break. A book cannot vanish from your hands mid-sentence, but a web page can be deleted and the link becomes a dead end. Berners-Lee chose one-way links precisely because two-way links need permission from both ends, and requiring permission would have stopped the whole thing growing.

30.10.3 PLAIN - a worked example

1. Take one address apart. Every part does one job.

```
https://info.cern.ch/hypertext/WWW/TheProject.html
\___/   \_________/\_____________________________/
scheme      host                path

GET /hypertext/WWW/TheProject.html HTTP/1.1
Host: info.cern.ch

HTTP/1.1 200 OK
Content-Type: text/html
```

2. The scheme says which protocol to speak, the host says which machine, and the path says which document on it.
3. The response starts with a number. 200 means here it is, 404 means no such document, 301 means it moved. The body is HTML: text with tags marking headings, paragraphs and links.

Invention	Job
URL	Name any document anywhere
HTTP	Ask for it and get an answer
HTML	Write it with links inside

4. None of the three was new on its own. The combination was: a global name, a trivial fetch, and a format that contains the next global name.

30.10.4 PLAIN - what is really happening inside

1. The web is an **application** that runs on the internet. It is not the internet.
2. The internet is the delivery system: addresses, routing, packets. Email, video calls, games, software updates, DNS and messaging apps all use the internet without using the web at all.
3. Three design choices made the web spread faster than its rivals.
4. **Links are one way.** You may link to any page without telling the owner, so the graph grows with no coordination.
5. **Broken links are tolerated.** A 404 is a normal answer, not a failure of the system. Designs that guaranteed link integrity could not scale.
6. **It was free of ownership.** CERN's 1993 release meant no company could charge for the protocol or block a competing browser.
7. The honest version: the web was not the best hypertext system designed. It was much weaker than several research systems of the 1980s. It won because it was the one that required permission from nobody.

30.10.5 TECHNICAL - the engineer's version

1. Protocol versions, with dates, because the differences matter in practice.

Version	Year	Key change
HTTP/0.9	1991	GET only, no headers
HTTP/1.0, RFC 1945	1996	Headers, status codes
HTTP/1.1, RFC 2068	1997	Persistent connections
HTTP/2, RFC 7540	2015	Multiplexing, binary
HTTP/3, RFC 9114	2022	Runs over QUIC and UDP

2. URLs were specified in RFC 1738 in 1994. The current generic URI syntax is RFC 3986 from 2005.
3. HTTP/3 uses QUIC (RFC 9000, 2021), which runs over UDP rather than TCP, because TCP's ordering guarantee causes head-of-line blocking when a single packet is lost.
4. To see the raw protocol yourself, run `curl -v` against any site. It prints the request and response headers exactly as described above.

30.10.6 WORDS - remember these

1. Web — pages with links that you read in a browser — a distributed hypertext application layered on the internet.
2. URL — the full address of one document — a Uniform Resource Locator, syntax defined in RFC 3986.
3. HTTP — the request and answer language — the stateless application protocol for retrieving and submitting resources.
4. HTML — the way pages are written — a markup language describing structure and hyperlinks, now a living standard with no version number.

30.11 The browser wars

30.11.1 PLAIN - in simple words

1. Early web browsers showed text and made you open images in a separate window.
2. In January 1993 Marc Andreessen and Eric Bina at the National Center for Supercomputing Applications released an alpha of a browser called Mosaic. Version 1.0 came on 21 April 1993.

3. Mosaic showed pictures inline with the text. That sounds small. It changed who wanted to use the web.
4. Andreessen left and, with Jim Clark, founded a company on 4 April 1994 that became Netscape. Its browser was released as Mosaic Netscape 0.9 on 13 October 1994 and took over the web almost immediately.
5. Netscape went public on 9 August 1995. Shares were offered at 14 dollars, doubled to 28 before trading opened, touched 75 on the first day and closed at 58.25, valuing a company with little revenue at about 2.9 billion dollars.
6. Microsoft responded by giving its browser away and putting it inside Windows. Netscape lost, was bought by AOL, and released its source code, which became Mozilla and eventually Firefox.
7. Then Google entered with Chrome in 2008 and won more completely than anyone before it.

30.11.2 PLAIN - a picture in your head

1. Imagine two companies selling door locks for a new kind of house.
2. Company B also sells houses, and starts fitting its own lock to every house it builds, free and already installed.
3. Company A cannot compete with free, or with pre-installed, and goes out of business. Later, a third company that owns the road everyone drives on offers a new lock and mentions it on every journey.

Where this comparison breaks: locks do not improve when more people use them, but browsers do. Popularity made web authors test against a browser, which made it work better, which made it more popular. That loop is why browser markets tip so hard toward one winner and are so difficult to reverse.

30.11.3 PLAIN - a worked example

1. Follow the market share and notice how fast each turn happened.

Period	Leader	Rough share
1995 to 1997	Netscape	Majority
2001 to 2004	Internet Explorer	Over 90 percent
2010	Firefox and IE	Near parity
July 2026	Chrome	68.22 percent

2. The July 2026 worldwide figures from Statcounter page-view measurement were Chrome 68.22, Safari 16.47, Edge 5.37, Firefox 3.34, Samsung Internet 2.06 and Opera 1.88 percent.
3. Be careful with such numbers. They measure page views from participating sites, not installed copies, and different measurement firms disagree by several points.
4. Edge and Opera are built on the same engine as Chrome, so engine share is far more concentrated than browser share.

30.11.4 PLAIN - what is really happening inside

1. A browser is several programs in one, and the important one is the **rendering engine**.
2. The rendering engine takes HTML, CSS and images and works out exactly what pixels go where. Beside it sit a JavaScript engine and a networking layer.
3. Building a rendering engine is enormously expensive. It must handle every badly written page on the web and match the leaders bug for bug. That is why the number of independent engines keeps falling.
4. Three engines survive in general use in 2026.
5. **Blink**, used by Chrome, Edge, Opera, Brave and Samsung Internet, forked from WebKit by Google in 2013.
6. **WebKit**, used by Safari, forked by Apple from KHTML, from the KDE project, in 2001.

7. **Gecko**, used by Firefox, descends directly from the Netscape code released in 1998. So the code Netscape published when it lost is still running today, which is a fair result for the loser of the first browser war.

30.11.5 TECHNICAL - the engineer's version

1. NCSA Mosaic 0.5 alpha was announced on 23 January 1993, version 1.0 came on 21 April 1993, and NCSA ended support in January 1997.
2. Internet Explorer 1.0 shipped on 24 August 1995 in the Microsoft Plus pack for Windows 95. IE peaked above 90 percent usage share in 2001.
3. United States v. Microsoft was filed in 1998. The district court found a Sherman Act violation and in 2000 ordered a breakup; the appeals court reversed the breakup in 2001, and a settlement with conduct remedies rather than divestiture was approved in 2002.
4. Netscape announced on 22 January 1998 that its source would be free and developed openly as Mozilla. AOL announced the acquisition on 24 November 1998 at 4.2 billion dollars in stock, worth about 10 billion when it closed on 17 March 1999.
5. Firefox 1.0 was released 9 November 2004. Google Chrome was released 2 September 2008.

Engine	Origin	Used by
Gecko	Netscape code, 1998	Firefox
WebKit	KHTML fork, 2001	Safari
Blink	WebKit fork, 2013	Chrome, Edge, Opera

6. Microsoft retired its own Trident and EdgeHTML engines when Edge moved to Blink, released in January 2020. That took the world from four independent engines to three.
7. The concentration matters technically, not only commercially. When one engine has over two-thirds of usage, "it works in Chrome" starts to replace "it follows the specification" as the working definition of correct.
8. This is a live disagreement. One side argues consolidation makes the web more consistent and lets features ship faster. The other argues it hands one company effective control of a public standard. Serious people hold both positions.

30.11.6 WORDS - remember these

1. Browser — the program that shows web pages — a user agent implementing HTTP, HTML, CSS and scripting.
2. Rendering engine — the part that decides what the page looks like — the layout and paint engine turning markup and styles into pixels.
3. Bundling — shipping it already installed — including a product with an operating system, the core issue in the Microsoft antitrust case.

30.12 The dot-com boom and bust

30.12.1 PLAIN - in simple words

1. In 1995 three things happened at once and together they lit the fire.
2. Windows 95 put a usable computer in ordinary homes, Netscape made the web easy, and the NSFNET ban on commercial traffic was gone.
3. Suddenly anyone could sell anything to anyone, and nobody knew how big that would be.
4. Netscape's listing on 9 August 1995 taught investors a lesson they took too far: a company with almost no profit could be worth billions.
5. Money poured in. The accepted wisdom was "get big fast": collect users now and worry about profit later. Users were called eyeballs.

6. The NASDAQ Composite index peaked on 10 March 2000 at 5,048.62, then fell to about 1,114 on 9 October 2002, a drop of roughly 78 percent.
7. Roughly 5 trillion US dollars of market value was destroyed by the end of 2002. Some companies survived and became the largest in the world. Most did not.

30.12.2 PLAIN - a picture in your head

1. Imagine a gold rush in a valley where gold has genuinely been found.
2. The gold is real. That matters. This was not a fantasy.
3. Eventually money stops arriving, and the diggers with no gold cannot pay wages and stop within weeks. The few who found real seams keep working and in twenty years own the valley.

Where this comparison breaks: gold is finite and the internet market was not. The survivors were building businesses whose costs fell as they grew, which is the opposite of mining. The bubble was in the pricing, not in the underlying opportunity, and telling those two apart is the hardest judgement in investing.

30.12.3 PLAIN - a worked example

1. Pets.com is the standard example. Founded November 1998, it ran a Super Bowl advertisement in January 2000 costing about 1.2 million dollars, went public in February 2000 at 11 dollars per share raising 82.5 million, and shut down on 9 November 2000, about nine months later.
2. The fatal detail: shipping heavy pet food cost more than the margin on it, so every sale lost money and growth made the loss bigger.
3. Webvan, founded 1996, raised 375 million dollars at its November 1999 listing and reached a valuation over 4.8 billion. In 2000 it had 178.5 million dollars of sales against 525.4 million of expenses, and filed for bankruptcy in July 2001 having lost over 800 million.
4. Boo.com, founded 1998 in London, spent 135 million dollars of venture capital in about 18 months and went into receivership on 18 May 2000.

Company	Founded	Outcome
Amazon	1994	Survived, now huge
eBay	1995	Survived, profitable early
Google	1998	Survived, dominant
Cisco	1984	Survived, lost 80 pct
Netscape	1994	Sold to AOL, 1999
Webvan	1996	Bankrupt July 2001
Boo.com	1998	Receivership May 2000
Pets.com	1998	Closed 9 Nov 2000

5. Amazon's share price fell around 90 percent from its peak and it survived anyway, largely because it had raised bond money before the market closed.
6. That is the uncomfortable truth: survival depended heavily on how much cash was in the bank when the funding stopped.

30.12.4 PLAIN - what is really happening inside

1. Separate four things that get blended together in the retelling.
2. **The technology was real.** Internet use grew right through the crash and never stopped.
3. **The business models often were not.** Free delivery of heavy goods at a loss does not become profitable at scale.
4. **The valuations were unmoored.** Companies were priced on visitor counts because there were no earnings to price on.

5. **The financing was fragile.** Most needed a new funding round every year, so when funding stopped they died within months.
6. Alan Greenspan used the phrase "irrational exuberance" about asset prices in December 1996, more than three years before the peak. Being early is the same as being wrong, until it is not.
7. The honest lesson is not that the internet was overhyped. Amazon in 2026 is larger than any 1999 forecast. The forecasts were not too optimistic about the internet. They were too optimistic about who would capture the value, and how soon.

30.12.5 TECHNICAL - the engineer's version

1. NASDAQ Composite closing peak: 5,048.62 on 10 March 2000. Trough: about 1,114 on 9 October 2002, roughly 78 percent below the peak, giving up all the gains of the bubble.
2. From 1995 to March 2000 the index rose about 600 percent. In 1999 alone it rose 85.6 percent, while more listed stocks fell than rose.
3. The index did not regain its March 2000 closing level until 2015.
4. The infrastructure overbuild had a long tail. Enormous amounts of optical fibre were laid and left unlit. That dark fibre was bought cheaply later and is part of why bandwidth prices fell so far in the 2000s.
5. WorldCom, a major carrier of the era, collapsed in an accounting fraud, was renamed MCI in 2003 and acquired by Verizon in 2006.
6. A useful measure for judging any such period is burn rate: cash spent per month, divided into cash on hand, gives runway in months. Most failed dot-coms had a runway under 12 months at all times.
7. The pattern generalizes. Real technology, unclear business models and stretched valuations appear in later cycles too, and reasonable people disagree in 2026 about whether current AI spending is a repeat, an overshoot, or neither. Treat that as an open argument, not a settled fact.

30.12.6 WORDS - remember these

1. Eyeballs — a count of visitors used instead of profit — audience metrics used as a valuation proxy when there are no earnings.
2. Burn rate — how fast the money is spent — net cash outflow per month, which divided into reserves gives runway.
3. Dark fibre — cable laid but not switched on — installed optical fibre with no active transmission equipment attached.

30.13 Broadband and the physical internet

30.13.1 PLAIN - in simple words

1. For most of the 1990s, home internet came through a modem over an ordinary phone line. A modem turns bits into sounds, because the phone network only carried sound.
2. That is why dial-up made a noise. The two modems were negotiating out loud: testing the line, agreeing a speed and measuring the echo.
3. The noise stopped once agreement was reached, because the speaker was switched off, not because the sounds stopped.
4. Then came ways to use the same copper without occupying the voice channel, which is what ADSL does. ADSL gives more download than upload speed on purpose, which is what the A means.
5. Cable television networks were adapted to carry data, and eventually fibre was run all the way into homes.
6. Underneath all of it sits the part almost nobody sees: over a million kilometres of cable on the sea floor, carrying about 99 percent of intercontinental data. Satellites carry a tiny fraction.

30.13.2 PLAIN - a picture in your head

1. Think of the phone line into a house as a wide pipe that was only ever used at one end.
2. Voice needs the frequencies a human ear cares about, roughly 300 to 3,400 cycles per second. The copper can carry far more than that.
3. Dial-up squeezed data into the voice part of the pipe, so it competed with speech and was capped near 56 thousand bits per second.
4. ADSL puts data in the unused higher frequencies and leaves the voice part alone, with a small filter at each phone keeping them apart.
5. The download side gets the bigger share because that is what people mostly do, and because upstream signals are weakest exactly where interference is worst. Fibre replaces the pipe with glass, where the usable range is thousands of times wider.

Where this comparison breaks: a water pipe's capacity does not fall with length, but copper's does, badly. ADSL speed drops steadily with distance from the exchange and is usually unusable past about 4 to 5 km. Fibre does not care about a few kilometres, which is why the industry moved to it rather than squeezing copper further.

30.13.3 PLAIN - a worked example

1. Follow one home connection through its generations.

Technology	Typical down	Typical up
V.34 modem, 1994	28.8 kbit/s	28.8 kbit/s
V.90 modem, 1998	56 kbit/s	33.6 kbit/s
ISDN basic rate	128 kbit/s	128 kbit/s
ADSL2+	Up to 24 Mbit/s	About 1 Mbit/s
GPON fibre	100 to 1000 Mbit/s	20 to 1000 Mbit/s

2. Take a 4-megabyte photograph, which is 32 million bits.
3. On a V.90 modem at 56 kbit/s that is about 570 seconds, so nearly ten minutes. On ADSL2+ at 20 Mbit/s it is about 1.6 seconds. On a 300 Mbit/s fibre link it is about 0.1 seconds.
4. Now the ocean. A single modern transatlantic cable such as MAREA is rated at about 224 terabits per second.
5. That one cable could move that photograph roughly 7 million times per second, which is why cables and not satellites carry the world's traffic.

30.13.4 PLAIN - what is really happening inside

1. A submarine cable is thinner than you expect: about 25 mm across in deep water, weighing roughly 1.4 tonnes per kilometre.
2. Inside are a few pairs of glass fibres, a steel strength member, a copper conductor and layers of insulation and water barrier.
3. The copper is not for data. It carries direct current at several thousand volts to power repeaters on the sea floor, roughly every 60 to 100 km.
4. A cable ship lays it slowly, and in shallow water a plough buries it about a metre under the seabed to protect it from anchors and trawlers.
5. Cables break often. About 200 faults happen worldwide each year, and about two-thirds are caused by fishing gear and ships' anchors.
6. Repair means sending a ship, grappling the cable up, cutting it, splicing in a new length and lowering it back. That takes days to weeks depending on weather, permits and ship availability, which is why redundancy matters more than repair speed.

7. Satellites are different. A geostationary satellite sits about 35,786 km above the equator, so light takes about 240 milliseconds for the round trip before any processing. Low earth orbit constellations fly around 550 km up, which cuts that delay enormously at the price of needing thousands of satellites.

30.13.5 TECHNICAL - the engineer's version

1. Modem standards, with years: Bell 103 at 300 bit/s in 1962; V.22bis at 2,400 in 1984; V.32 at 9,600 in 1984; V.32bis at 14,400 in 1991; V.34 at 28.8 then 33.6 kbit/s from 1994 to 1996; V.90 at 56 down and 33.6 up in 1998; V.92 in 2000, raising upload toward 48 kbit/s.
2. The dial-up handshake is a defined training sequence: answer tone, echo canceller disabling tone, probing signals to measure the channel, then rate negotiation. It is audible only because the monitor speaker is on by default, controlled by the `ATM` and `ATL` commands.
3. ADSL, ITU-T G.992.1, uses discrete multitone modulation. In Annex A, 26.075 to 137.825 kHz carries upstream and 138 to 1104 kHz downstream. ADSL2+ is G.992.5, reaching about 24 Mbit/s down. VDSL2 is G.993.2.
4. Asymmetry has two causes: crosstalk is worst at the exchange end where the upstream signal is already weakest, and demand is genuinely asymmetric. The first is engineering, the second is marketing. Both are real.
5. ISDN basic rate is 2B+D: two 64 kbit/s bearer channels plus a 16 kbit/s signalling channel, so 128 kbit/s bonded.
6. Submarine facts as of early 2026: more than 600 active and planned cables, over 1.5 million km in service, carrying about 99 percent of international data traffic. Fault rates fell from about 3.7 per 1,000 km per year between 1959 and 1979 to about 0.44 after burial became standard in 1985.

Orbit	Altitude	Typical round trip
Geostationary	35,786 km	About 240 to 600 ms
Low earth orbit	About 550 km	About 25 to 35 ms
Terrestrial fibre	Ground	About 1 ms per 100 km

7. Starlink reported approximately 10,400 satellites in orbit and more than 12 million subscribers as of June 2026, with typical latency around 25 to 35 ms.
8. India specifically. Public internet began on 15 August 1995, provided by Videsh Sanchar Nigam Limited. TRAI reported about 918 million internet subscribers as of 30 September 2023, of which about 881 million were wireless and about 37 million wired.
9. India is served by roughly 17 international submarine cables landing at about 5 main landing points, chiefly Mumbai and Chennai, including SEA-ME-WE 3, Bay of Bengal Gateway and Europe India Gateway. Reliance completed the IAX and IEX systems in 2024.
10. That concentration is the real chokepoint. A small number of landing stations in two cities serves 1.4 billion people, and cable damage in the Red Sea and near Egypt has repeatedly degraded Indian and Gulf connectivity because so many routes share one corridor.
11. The reader's own traceroute shows this geography directly: Delhi, then Mumbai, then Pune, all inside Microsoft's backbone, because that is where the capacity and the landing stations are.
12. Speedtest Global Index figures for India in November 2023 were about 94.62 Mbit/s mobile download and 58.62 Mbit/s fixed download. Mobile being faster than fixed is unusual worldwide and reflects how India built out.

30.13.6 WORDS - remember these

1. Modem — the box that turned bits into sound — a modulator and demodulator converting digital data to and from an analogue channel.
2. ADSL — fast down, slower up, over phone copper — asymmetric digital subscriber line, ITU-T G.992 series, using discrete multitone modulation.

3. Repeater — the thing on the sea floor that keeps the light strong — an optical amplifier, usually erbium-doped, powered over the cable's copper.
4. Landing station — where a sea cable comes ashore — the terminal facility that powers the cable and connects it to terrestrial networks.
5. Chokepoint — a place where too much depends on too little — a corridor or facility whose failure degrades connectivity for a whole region.

30.14 Who actually runs the internet

30.14.1 PLAIN - in simple words

1. Nobody owns the internet, and that is not a slogan. There is no company or government that can switch it off.
2. But some things must be agreed globally or nothing works.
3. Names must be unique, so two people cannot own the same domain. Addresses must be unique, so a packet has one destination. Protocols must be shared, so machines built by strangers can talk.
4. Three kinds of body handle those three jobs, and none controls the network.
5. ICANN coordinates names and, through IANA, the address and number pools.
6. The IETF writes the protocol specifications, published as RFCs. The W3C writes web standards, and five regional registries hand out address blocks in their own parts of the world.
7. Everything else is contracts between companies, and national law.

30.14.2 PLAIN - a picture in your head

1. Think of the world's postal systems. No single body runs them.
2. Someone must agree country codes, or letters cannot be sorted. Someone must agree envelope sizes and address layout so machines can read them. Small, boring, essential jobs.
3. The internet has the same shape: a tiny agreed core and an enormous uncoordinated periphery.

Where this comparison breaks: postal agreements are made between governments in a treaty organization. Internet standards are made largely by engineers and companies, with governments as one voice among several. That is the multistakeholder model, and several governments have objected to it for decades, wanting state control instead. The argument is unresolved.

30.14.3 PLAIN - a worked example

1. Here is how a proposal actually becomes an internet standard.
2. Someone writes an **Internet-Draft** and posts it. Anyone may do this, with no membership required.
3. A working group discusses it on a public mailing list, archived and readable by anyone. Implementations are expected, because working code carries more weight than argument.
4. When the group reaches **rough consensus**, meaning no serious unanswered objection rather than a majority vote, it goes to the steering group.
5. If approved it is published as an RFC with the status Proposed Standard. If it proves itself in deployment it can be raised to Internet Standard, and most useful protocols never bother.

```
Idea -> Internet-Draft -> working group list ->
running code -> rough consensus -> IESG review ->
RFC (Proposed Standard) -> deployment ->
Internet Standard
```

6. Meetings famously test consensus by humming rather than voting, so volume rather than headcount is judged and nobody is formally counted.

30.14.4 PLAIN – what is really happening inside

1. Keep the layers of governance apart, because they are often confused.
2. **Unique identifiers.** ICANN coordinates the domain name root and the number pools, and decides which top-level domains exist.
3. **Protocol specifications.** The IETF decides how TCP, HTTP and DNS behave. It has no power over who uses them.
4. **Everything else.** Which cables exist, who peers with whom, what content is allowed and what is censored is decided by companies and states.
5. That is why "who runs the internet" has an uncomfortable answer: for the things people care about most, it is your government and your internet provider. Net neutrality is exactly the argument about that last point.

30.14.5 TECHNICAL – the engineer's version

1. ICANN was incorporated in California on 30 September 1998, taking over functions previously performed by Jon Postel at USC-ISI. It does not control content, spam, or what any network carries.
2. The IANA functions contract with the US National Telecommunications and Information Administration ended on 1 October 2016, transferring stewardship to the global multistakeholder community.
3. The IETF held its first meeting on 16 January 1986, with 21 US federal-government-funded researchers attending.
4. RFC 1 was published on 7 April 1969 by Steve Crocker of UCLA, titled "Host Software". Jon Postel edited the series from 1969 until his death in 1998.
5. The process is defined in RFC 2026, "The Internet Standards Process, Revision 3". RFC 6410, October 2011, cut the maturity ladder from three levels to two by retiring Draft Standard.
6. "Rough consensus and running code" comes from David Clark's 1992 IETF presentation, where he said the group rejects kings, presidents and voting, and believes in rough consensus and running code.

Registry	Region	Since
RIPE NCC	Europe, West Asia	1992
APNIC	Asia Pacific, India	1993
ARIN	North America	1997
LACNIC	Latin America	1999
AFRINIC	Africa	2004

7. The IANA free pool of IPv4 addresses was exhausted on 3 February 2011, when the last five blocks went to the five registries. APNIC, which covers India, effectively ran out in April 2011, earliest of all.
8. The W3C was founded in October 1994 by Tim Berners-Lee at MIT. Since 2019 the HTML and DOM specifications are maintained by the WHATWG with W3C endorsement, after years of the two bodies publishing rival versions.
9. Net neutrality in the United States, in dates: the FCC voted for Title II common-carrier treatment on 26 February 2015, effective 12 June 2015; repealed it on 14 December 2017, effective 11 June 2018; voted to restore it on 25 April 2024; and the Sixth Circuit struck that down in January 2025. As of 2026 there is no federal rule, though state laws such as California's SB 822, signed 30 September 2018, remain.
10. Both sides, stated fairly. For: without a rule a provider can slow or charge for competing services, and small entrants cannot negotiate. Against: broadband needs heavy investment, traffic genuinely differs in requirements, and common-carrier rules designed for telephones discourage building. Serious economists hold both views.
11. India took a strong position. TRAI prohibited discriminatory pricing of data services in February 2016, ending zero-rating schemes, and recommended full net neutrality rules in November 2017.

30.14.6 WORDS - remember these

1. ICANN — the body that coordinates names and numbers — the corporation overseeing the DNS root, gTLD policy and the IANA functions.
2. RFC — the document a standard is written in — a numbered publication of the RFC series, of which only some are standards-track.
3. Rough consensus — no serious unanswered objection — the IETF decision rule, explicitly not a majority vote.
4. Net neutrality — the rule that providers must not favour some traffic — the principle that carriers should not block, throttle or paid-prioritize.

30.15 Web 1.0 to now

30.15.1 PLAIN - in simple words

1. The first web was files. Someone wrote an HTML file, put it on a server, and everyone who asked got the same file.
2. Then pages started being built when you asked for them, from a database, so a shop could show your basket rather than a fixed page.
3. From about 2004 the phrase Web 2.0 described sites where the users made the content: blogs, wikis, photos, video and comments.
4. Social networks turned that into the main use of the web, and smartphones changed it again. From around 2016 more web traffic came from phones than from computers.
5. Apps took a large share of that time back from the web, inside private platforms with their own rules.
6. Running your own server became rare, and most services now run on cloud infrastructure owned by a handful of companies.
7. So traffic and attention both concentrated, on a network originally designed to have no centre.

30.15.2 PLAIN - a picture in your head

1. Imagine a town that starts as thousands of small shops, each owner-built.
2. Then a few large market halls open. They are cleaner, easier and free to enter, so almost everyone moves inside.
3. The streets still exist and the roads are still public, but nearly all the trade happens in five buildings, and each landlord sets the rules inside.
4. Nobody forced anyone. Each move was individually sensible.

Where this comparison breaks: a shop that leaves a market hall can reopen on the street. On the web, leaving usually means losing the audience, because the audience is a feature of the hall rather than of the shop. That lock-in has no clean equivalent in a market town.

30.15.3 PLAIN - a worked example

1. Compare the same task across eras: publishing a photograph for friends.

Era	Method	Who controls it
1996	Edit HTML, upload	You
2003	Blog software	You, mostly
2010	Social network post	The platform
2026	App feed post	The platform

2. In 2026 anyone can do it in four seconds, and cannot control who sees it, how long it lasts, or whether it is shown at all.
3. That is the trade in one table: an enormous gain in access, and a real loss of control.

30.15.4 PLAIN - what is really happening inside

1. Say plainly what was gained. Publishing became free and instant for billions of people who had no voice before. Search made the whole archive findable. Encryption became normal, so most traffic is private in transit.
2. Say plainly what was lost. Independence: most content now lives on servers the author does not control.
3. Durability: pages vanish when a platform closes, and nobody is obliged to keep them.
4. Interoperability: an early web page could be read by any client, and a modern app feed usually cannot.
5. Privacy in a different sense: the cost of free services is measurement of behaviour at a scale nobody imagined in 1993.
6. Resilience: a design meant to have no single point of failure now leans heavily on a few cloud and content-delivery providers, so single configuration errors have caused worldwide outages.
7. None of that was forced by the technology. It followed from economics: scale is cheaper, and defaults decide behaviour.

30.15.5 TECHNICAL - the engineer's version

1. The phases in technical terms: static file serving; then server-side generation with CGI (RFC 3875) and later application servers; then asynchronous updates in the browser using XMLHttpRequest, which made single-page applications practical from around 2005.
2. The term Web 2.0 was popularized by the O'Reilly Media conference of that name held from 2004. It described a business and design pattern, not a protocol version. There was never a Web 1.0 specification.
3. Mobile passed desktop in worldwide web page views around late 2016 by Statcounter's measurement, and has stayed the majority since.
4. Encryption in transit became the default rather than the exception. Let's Encrypt began issuing free certificates in 2015, and the share of page loads over HTTPS in Firefox telemetry rose from under 30 percent in 2014 to above 80 percent by 2020.
5. Content delivery networks now terminate a large share of connections close to users, so your packets often never reach the origin server's country. That is exactly what the reader's trace shows, ending inside Microsoft's own network rather than at a distant data centre.
6. The honest assessment: the internet's core design remains decentralized and works. The application layer above it has centralized heavily. Those are two separate facts, often wrongly merged into one complaint.

30.15.6 WORDS - remember these

1. Web 2.0 — sites built from what users post — a design and business pattern from around 2004, not a technical standard.
2. Content delivery network — copies of a site kept near you — distributed caching and connection termination close to end users.
3. Lock-in — hard to leave without losing something — switching costs created by data, audience or formats that are not portable.

30.16 A master internet timeline

1. One table, from the semaphore to today. Every date here is discussed somewhere in this chapter.

Year	Event	Significance
1792	Chappe line approved	State message network
1794	Paris to Lille opens	Minutes instead of days

Year	Event	Significance
1837	Cooke-Wheatstone patent	Electric telegraph starts
1844	Washington to Baltimore	Morse line proves itself
1858	First Atlantic cable	Works 3 weeks, then dies
1866	Atlantic cable works	Continents linked for good
1874	Baudot and quadruplex	Multiplexing arrives
1876	Bell patent 174,465	Voice over wire
1891	Strowger patent 447,918	Automatic switching
1892	La Porte exchange opens	Operators replaced
1960	Baran begins at RAND	Distributed network idea
1962	Licklider heads IPTO	Networked computing aim
1963	Intergalactic memo	The aim written down
1964	Baran RM-3420 series	Message blocks published
1965	Davies designs at NPL	The word packet coined
1966	Taylor starts ARPANET	Project actually funded
1969	RFC 1 published	The RFC series begins
1969	First message, 29 Oct	Link goes live, then “LO”
1970	NPL network in service	UK packet network runs
1971	Tomlinson network email	The @ sign is chosen
1972	ICCC demonstration	Doubters convinced
1973	Norway and UK links	ARPANET goes global
1974	Cerf and Kahn paper	Internetworking defined
1976	X.25 published	The carriers’ rival
1978	TCP and IP split	The thin waist appears
1980	Usenet established	Discussion with no centre
1981	BITNET founded	Universities outside ARPA
1982	Minitel commercial	France online early
1983	Flag day, 1 January	ARPANET moves to TCP/IP
1983	4.2BSD ships TCP/IP	Free running code spreads
1984	DNS specified	Names scale past a file
1984	FidoNet created	Hobbyist global network
1984	ISO 7498 published	OSI reference model
1986	NSFNET at 56 kbit/s	Research backbone starts
1986	First IETF meeting	Standards body forms
1988	NSFNET T-1 backbone	1.544 Mbit/s core
1989	Berners-Lee proposal	Web idea written down
1990	ARPANET decommissioned	The original retires
1990	First web server runs	info.cern.ch goes live
1991	Web released publicly	Anyone can join in
1991	CIX formed	Commercial peering begins
1991	NSFNET T-3 backbone	45 Mbit/s core
1993	Mosaic released	Images inline, web grows
1993	CERN public domain	No fee, no permission
1994	Netscape founded	Web becomes an industry
1994	W3C founded	Web standards body
1995	Netscape IPO, 9 Aug	Boom psychology begins
1995	NSFNET off, 30 April	Internet goes commercial
1995	India public internet	VSNL service, 15 August
1995	IE 1.0 with Windows 95	Browser war opens
1998	ICANN incorporated	Names and numbers body

Year	Event	Significance
1998	Google founded	Search is the front door
1998	Mozilla source released	Gecko lineage begins
2000	NASDAQ peak, 10 March	5,048.62, then the fall
2001	IE above 90 percent	First browser war ends
2002	NASDAQ low, October	About 78 percent down
2003	NIXI registered	Indian traffic stays local
2004	Firefox 1.0	Competition returns
2008	Chrome released	Second browser war
2011	IPv4 pool exhausted	3 February, IANA dry
2012	Minitel switched off	30 June, closed model ends
2015	HTTP/2 standardized	Multiplexed web transport
2016	IANA transition	US contract ends, 1 Oct
2016	Mobile passes desktop	Phone-first web
2020	Edge moves to Blink	Three engines left
2022	HTTP/3 standardized	Web moves onto QUIC
2025	DE-CIX peak 18.73 Tbit/s	Scale of one exchange
2026	Over 600 sea cables	1.5 million km in service

30.98 Common wrong ideas

1. Wrong: the internet was built to survive a nuclear war. Right: Paul Baran's RAND work had a survivability motive, but ARPA's director stated plainly that the ARPANET was not started for that. It was built for resource sharing among scarce research computers.
2. Wrong: Al Gore claimed he invented the internet. Right: in a 1999 interview he said he took the initiative in creating the internet, meaning legislative and funding support. He sponsored the High Performance Computing Act of 1991, and Vint Cerf and Bob Kahn publicly defended his record. He never claimed to have invented it.
3. Wrong: the internet and the web are the same thing. Right: the internet is the network. The web is one application on it, created eighteen years later. Email, video calls and app traffic use the internet without the web.
4. Wrong: the internet is wireless now. Right: your last few metres may be wireless, but traffic goes onto fibre almost immediately, and about 99 percent of international traffic runs through cables on the sea floor.
5. Wrong: satellites carry most long-distance internet traffic. Right: they carry a very small fraction. Cables are far cheaper, far faster and far lower in delay.
6. Wrong: one organization runs the internet and could switch it off. Right: ICANN coordinates names and numbers, the IETF writes specifications, and nobody operates the whole network. Governments can cut off their own country, which is a different thing.
7. Wrong: Tim Berners-Lee invented hypertext. Right: hypertext ideas date to Vannevar Bush in 1945 and Ted Nelson in the 1960s. He combined URLs, HTTP and HTML into something anyone could deploy without permission.
8. Wrong: the dot-com crash proved the internet was hype. Right: usage kept growing right through the crash. The pricing and the business models were wrong, not the direction.
9. Wrong: Leonard Kleinrock invented packet switching. Right: he did important queueing analysis and later ARPANET performance work, but primary sources and historians credit Baran and Davies with the independent invention, and Kleinrock acknowledged in 2023 that his early published work was about message switching.
10. Wrong: TCP/IP won because it was technically the best design. Right: it won because it existed, worked, cost nothing, shipped in Berkeley Unix, and had a guaranteed government market while OSI was still

being drafted.

30.99 Chapter summary in 20 lines

1. Long-distance messaging began by making a signal visible from the next hill, then giving it parts so it could spell anything.
2. Chappe's towers, from 1792, cut Paris to Lille from two days to about 32 minutes, and introduced encoding, relaying and a shared code book.
3. Electricity removed the need for line of sight. Cooke and Wheatstone patented in May 1837 and Morse's line ran on 24 May 1844.
4. The 1858 Atlantic cable managed about a word every ten minutes and died in weeks. The 1866 cable worked and shrank the world permanently.
5. The telephone, patented by Bell on 7 March 1876, needed exchanges, and Strowger automated them from 1891.
6. Circuit switching books a path for a whole call. That suits speech and wastes almost everything on a bursty computer conversation.
7. Packet switching splits messages into addressed pieces that travel independently and are reassembled at the far end.
8. It was invented three times: Baran at RAND from 1960 with a survivability motive, Davies at NPL in 1965 who named the packet, and Kleinrock supplying queueing analysis.
9. The ARPANET was built for resource sharing, not nuclear survival. Its first message, on 29 October 1969, was "LO" before the far end crashed.
10. The IMP was the first router, and its key idea was that hosts never had to agree with each other, only with the box beside them.
11. Email was not in the plan. Tomlinson joined two programs in 1971, chose the @ sign, and messaging became the network's main use.
12. Cerf and Kahn's May 1974 paper solved internetworking by putting a common protocol over dissimilar networks instead of making them agree.
13. TCP and IP were split in 1978, giving a thin waist: many technologies below, many applications above, one narrow protocol between.
14. On 1 January 1983 the ARPANET switched to TCP/IP in a single flag day. Nothing that size can be done that way again, which is why IPv6 has taken thirty years.
15. TCP/IP beat X.25, Minitel, private online services and OSI because it had running code, no fees, US government demand and simplicity.
16. NSFNET carried the research internet from 1986, banned commercial traffic, then lifted the ban and shut down on 30 April 1995, leaving a mesh of commercial networks and exchange points.
17. The web was proposed in March 1989, working by the end of 1990, and put into the public domain on 30 April 1993. It is an application on the internet, not the internet.
18. Mosaic in 1993 and Netscape's 9 August 1995 listing started both the browser wars and the dot-com boom. The NASDAQ peaked on 10 March 2000 at 5,048.62 and fell about 78 percent.
19. Physically the internet is fibre: over 600 submarine cables and more than 1.5 million kilometres, carrying about 99 percent of international traffic, with India served by a handful of landing points.
20. The core stayed decentralized and the applications did not, which is the central tension of the internet in 2026.

31

CHAPTER

KB

Wireless - WiFi, Bluetooth, Cellular and GPS

31.0 What this chapter gives you

1. You will be able to explain what a radio wave is and why an antenna has the length it has, and work out the wavelength of any band in your head.
2. You will be able to explain how information is put onto a wave, from simple on-off keying up to the 4096-QAM used by WiFi 7.
3. You will be able to explain OFDM properly, and say why the same idea sits under WiFi, 4G LTE, 5G and digital television.
4. You will be able to name every WiFi standard from 1997 to 2024 with its year, its band and its top speed, and say what each one added.
5. You will be able to walk through what happens on the air and in software from the moment you click a network name to the moment data flows.
6. You will be able to explain Bluetooth classic and Bluetooth Low Energy as two different protocols that share a brand name, and say why.
7. You will be able to explain how a phone attaches to a cellular network, what the SIM card actually holds, and what each generation changed.
8. You will be able to explain how GPS works, including why four satellites are needed and why the satellites need a relativity correction.

31.1 Radio from first principles

31.1.1 PLAIN - in simple words

1. Push electricity back and forth along a piece of metal and it makes a wave that flies away into space. That is all radio is.
2. Electricity in a wire has two side effects. It makes a magnetic push around the wire, and an electric push along it.
3. If the electricity is steady, those pushes just sit there. If it changes, they change too.
4. A changing magnetic push creates an electric push, and a changing electric push creates a magnetic push. Each one keeps making the other.
5. So the pair can let go of the wire and keep going on their own, forever, at the speed of light.
6. That travelling pair is a **radio wave**. The metal that launches it is an **antenna**.
7. A second piece of metal far away feels the wave arrive, and the wave pushes electricity back and forth in it, very weakly. That wiggle is the signal.
8. **Frequency** is how many times per second the push reverses, measured in hertz. WiFi at 2.4 GHz reverses 2,400,000,000 times per second.
9. **Wavelength** is how far the wave travels during one full reversal. At 2.4 GHz that is about 12.3 centimetres.

31.1.2 PLAIN - a picture in your head

1. Think of a long rope tied to a post at the far end of a field.
2. You hold your end and flick your wrist. A bump travels to the post. You did not send the rope. You sent a disturbance in the rope.
3. Flick faster and the bumps are closer together. The bump's travel speed does not change, only the spacing.
4. That is exactly frequency and wavelength. Speed is fixed. Faster wiggling means shorter waves.

Where this comparison breaks: a rope needs to exist. A radio wave needs nothing. It crosses empty space with no material carrying it, and the Michelson-Morley experiment of 1887 found no carrying substance at all. Also, a rope wave is one wiggle. A radio wave is two wiggles at right angles, electric and magnetic, locked together.

31.1.3 PLAIN - a worked example

1. The rule is: speed equals frequency times wavelength, so lambda = c / f.
2. The speed of light, written c, is 299,792,458 metres per second.
3. WiFi channel 6 sits at 2.437 GHz. lambda = 299,792,458 / 2,437,000,000 = 0.1230 metres, about 12.3 cm.
4. WiFi channel 36 sits at 5.180 GHz. lambda = 0.0579 metres, about 5.8 cm.

Band centre	Wavelength	Quarter wave	Used by
900 MHz	33.3 cm	8.3 cm	LoRa, old cordless
2.437 GHz	12.3 cm	3.1 cm	WiFi, Bluetooth
5.180 GHz	5.8 cm	1.4 cm	WiFi 5 GHz
6.175 GHz	4.9 cm	1.2 cm	WiFi 6E and 7
28 GHz	1.07 cm	2.7 mm	5G millimetre wave

5. A wire radiates best when it is a simple fraction of the wavelength, usually one half or one quarter.
6. At 2.4 GHz a quarter wave is 3.1 cm, which is why the metal strip inside a router aerial is about three centimetres long.

31.1.4 PLAIN - what is really happening inside

1. Feed alternating current into the middle of a straight wire. Electrons pile up at one end, rush to the other, and back, at the frequency you feed in.
2. Very close to the wire the fields slosh back and forth and return their energy to the wire. This is the **near field**.
3. Further out, the field cannot get back before the wire has reversed. The loop pinches off and is stranded.
4. A stranded loop is a wave, and it must travel outwards at the speed of light. This is the **far field**.
5. The boundary is roughly one sixth of a wavelength, about 2 cm at 2.4 GHz. Inside, energy is stored. Outside, energy is radiated.
6. Length matters because the current wave bounces off each end of the wire.
7. If the wire is a half wavelength long, the reflection arrives back in step with the next push, and the wire rings like a tuning fork.
8. If the length is wrong, the reflection fights the next push, most power bounces back into the transmitter, and little is radiated.
9. That is why a broken aerial makes a radio go quiet and the transmitter run hot.

31.1.5 TECHNICAL - the engineer's version

1. Radio propagation is described by Maxwell's equations, published by James Clerk Maxwell in 1865. Heinrich Hertz demonstrated generation and detection of the waves between 1886 and 1888, which is why the unit is the hertz.

2. Radio waves and visible light are the same phenomenon at different frequencies. There is no physical boundary in the spectrum, only naming.
3. Photon energy is E = h x f, with h = 6.626 x 10^-34 joule seconds. At 2.4 GHz that is 1.59 x 10^-24 J, or 9.9 microelectronvolts.
4. Visible light photons carry about 2 electronvolts, roughly 200,000 times more, and breaking a chemical bond needs several electronvolts.
5. That is the physics behind calling radio **non-ionizing**: a 2.4 GHz photon cannot break a bond at any power level, only heat.

Region	Boundary	At 2.4 GHz
Reactive near field	r < lambda / 2pi	under 2.0 cm
Radiating near field	to 2 D^2 / lambda	depends on size
Far field	r > 2 D^2 / lambda	typically over 10 cm

6. Gain is focusing, not amplification. A 3 dBi antenna sends twice the power one way by sending less elsewhere.
7. Tools: a vector network analyser measures VSWR and impedance. In software, `iw dev wlan0 link` on Linux and `wdutil info` on macOS 13 and later report the received level of a live link.

31.1.6 WORDS - remember these

1. Wavelength — how long one wiggle is in space — lambda = c / f in metres.
2. Near field — the zone next to the antenna where energy sloshes — the reactive region within lambda / 2pi.
3. Far field — the zone where the wave has left and is travelling — the radiating region beyond 2 D^2 / lambda.
4. Non-ionizing — too weak per photon to break molecules — photon energy far below chemical bond energies, so only thermal effects.

31.2 Carrying information on a wave

31.2.1 PLAIN - in simple words

1. A plain wave with nothing done to it carries no information. It is a tone.
2. To carry information you must change the wave in a way the receiver can notice and undo. Changing it is called **modulation**.
3. There are only three things about a wave you can change: its height, called **amplitude**; its wiggle rate, called **frequency**; and its timing offset, called **phase**.
4. Each jump lands the wave in one of a fixed set of states. One state is a **symbol**.
5. Four possible states means each symbol carries two bits. Sixteen states means four bits, but sixteen states are harder to tell apart.
6. That trade, more bits per symbol against needing a cleaner signal, is the single most important idea in modern wireless.

31.2.2 PLAIN - a picture in your head

1. Imagine signalling to a friend across a valley with a lamp at night.
2. Brighter and dimmer is amplitude modulation. Swapping a red lamp for a green one is frequency modulation, because colour is the frequency of light.
3. Now agree four brightness levels: off, dim, medium, bright. Each flash carries two bits.

Where this comparison breaks: a lamp has one dimension, brightness. A radio symbol has two dimensions at once, because the in-phase and out-of-phase parts vary independently. That two-dimensional space is

what makes QAM possible. Also, real receivers use error-correcting codes that fix a certain number of wrong guesses per block, so failure is a cliff, not a slow fade.

31.2.3 PLAIN - a worked example

1. Here are the three classic modulations drawn as waves, reading the bits 1, 0, 1 from left to right.

```
Carrier alone (no information)
  /\  /\  /\  /\  /\  /\  /\  /\  /\
 /  \/  \/  \/  \/  \/  \/  \/  \/  \

AM / ASK : height changes, spacing fixed
  /\  /\    __  __    /\  /\
 /  \/  \  \/  \/  /  \/  \
   bit 1     bit 0     bit 1

FM / FSK : spacing changes, height fixed
  /\/\/\   /  \  /  \   /\/\/\
  \/\/\/   \  /  \  /   \/\/\/
   bit 1      bit 0        bit 1

PSK : height and spacing fixed, wave jumps in time
  /\  /\  |\/  \/  |\  /\
 /  \/  \ |  \/  \ | \/  \
          ^ flip    ^ flip
```

2. **ASK**, amplitude shift keying: two brightness levels. Morse code by radio is ASK.
3. **FSK**, frequency shift keying: two tones. Old dial-up modems and the Bluetooth radio use it.
4. **PSK**, phase shift keying: in step or half a cycle out of step. Two states, one bit per symbol. That case is **BPSK**.
5. **QPSK** uses four phase positions a quarter turn apart, two bits per symbol.
6. **QAM** varies phase and amplitude together, giving a grid of states.

Scheme	States	Bits per symbol	First WiFi use
BPSK	2	1	802.11a, 1999
QPSK	4	2	802.11a, 1999
16-QAM	16	4	802.11a, 1999
64-QAM	64	6	802.11a, 1999
256-QAM	256	8	802.11ac, 2013
1024-QAM	1024	10	802.11ax, 2019
4096-QAM	4096	12	802.11be, 2024

7. A **constellation diagram** pictures those states. The sideways axis is the in-phase part, the up axis is the out-of-phase part, and each allowed state is a dot.
8. BPSK is two dots, left and right. QPSK is four, one per corner. 16-QAM is a four-by-four grid, 256-QAM is sixteen by sixteen, 4096-QAM is sixty-four by sixty-four.
9. With noise, each received dot lands in a fuzzy cloud instead of exactly on its spot.
10. On a four-by-four grid the clouds have room. On a sixty-four by sixty-four grid they overlap and the receiver picks the wrong dot.

31.2.4 PLAIN - what is really happening inside

1. **Symbol rate** is how many states per second you send, limited by how much spectrum you may occupy.
2. **Bit rate** is symbol rate times bits per symbol, reduced by the error-correcting overhead.

3. So there are only two ways to go faster: more symbols per second, needing more spectrum, or more bits per symbol, needing a cleaner signal. There is no third way.
4. Spread spectrum looks like the opposite of going faster.
5. In **frequency hopping spread spectrum**, FHSS, the transmitter jumps between small channels many times a second in a pattern the receiver knows. Jamming one channel costs only the hops that landed there.
6. In **direct sequence spread spectrum**, DSSS, each bit is replaced by a fixed longer pattern of faster pulses called chips.
7. Sending eleven chips per bit spreads the signal over eleven times the bandwidth, so it looks weaker and wider.
8. The receiver multiplies by the same known pattern. The wanted signal adds up; noise, being unrelated, cancels. That gain is why GPS can be received from orbit despite arriving weaker than the background noise.
9. Now **OFDM**, which solves echo. Indoors a signal arrives several times: direct, off a wall, off the floor, late by tens of nanoseconds.
10. OFDM makes each symbol much longer, four microseconds instead of fifty nanoseconds, so an echo overlaps only a tiny fraction of it.
11. Long symbols mean a slow symbol rate, so OFDM sends many symbols at once, side by side, on many narrow subcarriers.
12. The subcarriers are spaced so that at the exact centre of each, every other one passes through zero. That is what **orthogonal** means here, and it lets them overlap in frequency without interfering.
13. Each symbol is sent with a copy of its own tail glued to the front, the guard interval or cyclic prefix. Echoes land in the copy, which is thrown away.

31.2.5 TECHNICAL - the engineer's version

1. Frequency hopping was patented by Hedy Lamarr, filed under her legal name Hedy Kiesler Markey, with the composer George Antheil. US Patent 2,292,387, "Secret Communication System", was granted on 11 August 1942.
2. The mechanism used a punched roll like a player piano to step a torpedo guidance link through frequencies so it could not be jammed.
3. The US Navy rejected it as too bulky, and the patent expired before similar systems entered service around 1962.
4. It is honest to say Lamarr and Antheil did not invent WiFi. They patented one idea, decades before the electronics existed to do it cheaply.
5. Shannon's capacity theorem, published by Claude Shannon in 1948, sets the ceiling: $C = B \times \log2(1 + S/N)$.
6. Worked example: a 20 MHz channel at 25 dB SNR. 25 dB is a ratio of 316. $C = 20{,}000{,}000 \times \log2(317) = 166$ Mbit/s.
7. That is why one 20 MHz stream cannot exceed roughly 170 Mbit/s; real 802.11ax tops out at 143.4 Mbit/s.
8. 802.11b DSSS: 1 and 2 Mbit/s use an 11-chip Barker sequence at 11 Mchip/s. 5.5 and 11 Mbit/s use Complementary Code Keying with 8-chip codewords.

Parameter	802.11a/g value
FFT size	64 points
Subcarrier spacing	312.5 kHz
Data subcarriers	48
Pilot subcarriers	4
Symbol time	3.2 us plus 0.8 us guard

9. Check the top rate: 48 subcarriers x 6 bits (64-QAM) x 3/4 coding = 216 bits per symbol, divided by 4.0 microseconds = 54 Mbit/s exactly.
10. 802.11ax quadrupled the symbol: 78.125 kHz spacing, 256-point FFT for 20 MHz, 234 data subcarriers, 12.8 us symbol plus a 0.8, 1.6 or 3.2 us guard.
11. The longer symbol tolerates longer echoes, which is why 802.11ax works better outdoors and in large halls.

Modulation and coding	Rate, 1 stream	Needed SNR
BPSK 1/2 (MCS0)	8.6 Mbit/s	about 2 dB
16-QAM 1/2 (MCS3)	34.4 Mbit/s	about 11 dB
64-QAM 3/4 (MCS6)	68.8 Mbit/s	about 22 dB
1024-QAM 5/6 (MCS11)	143.4 Mbit/s	about 35 dB

12. Those SNR values are typical receiver sensitivity figures published by chip vendors, not values written into the standard.

31.2.6 WORDS - remember these

1. Modulation — changing a wave so it carries information — varying amplitude, frequency or phase of a carrier according to a message.
2. QAM — packing bits by changing both strength and timing — quadrature amplitude modulation with independent I and Q levels.
3. Spread spectrum — deliberately using more bandwidth than needed — trading bandwidth for processing gain and jam resistance.
4. DSSS — every bit sent as a longer known pattern — direct sequence spreading by a chip code such as the 11-chip Barker sequence.
5. FHSS — jumping between channels on a shared schedule — frequency hopping, 1600 hops per second in Bluetooth.
6. OFDM — many slow narrow signals instead of one fast wide one — orthogonal frequency division multiplexing with IFFT/FFT and a cyclic prefix.

31.3 The realities of radio

31.3.1 PLAIN - in simple words

1. A radio link is a shouting match across a noisy room. Three things decide whether it works: how loud you shout, how much the journey swallows, and how noisy the room already is.
2. Radio power is measured in **dBm**. 0 dBm means one milliwatt. Every 10 dBm added multiplies power by ten; every 10 dBm removed divides by ten.
3. A router transmits at about +20 dBm, which is 100 milliwatts. Your laptop hears it at about -50 dBm, one hundred billionth of a milliwatt.
4. The room is never silent. There is always a hiss from heat in the electronics and from the universe. That hiss is the **noise floor**.
5. What matters is not how loud your signal is, but how far above the hiss it is. That gap is the **signal-to-noise ratio**.
6. Signal also arrives more than once, bouncing off walls. The copies can add up or cancel out. That is **multipath**.
7. Lower frequencies bend around obstacles and pass through walls better, while higher ones carry more data but are stopped more easily.
8. That is the whole 2.4 versus 5 GHz story: 2.4 goes further, 5 goes faster.

31.3.2 PLAIN - a picture in your head

1. Imagine a party. Your voice loudness is the transmit power. The distance and the bodies between you are the path loss. Everyone else talking is the noise floor.
2. Add echo. Your voice arrives directly and again off the wall behind them. A short echo is unnoticed; a long one smears words together.
3. Now imagine you and a friend speaking at once from different places, each saying half a sentence. The listener has two ears and can separate you by direction. That is MIMO.

Where this comparison breaks: at a party, two people talking at once is always worse. In radio, two transmitters and two receivers with enough echo in the room can genuinely double the data, because the echoes make the paths mathematically separable. Multipath, the enemy in older systems, became the resource MIMO exploits.

31.3.3 PLAIN - a worked example

1. A real link budget for a laptop five metres from a router.
2. Router transmit power +20 dBm, plus 3 dBi antenna gain, gives +23 dBm radiated.
3. Free space loss at 5 metres and 2.4 GHz is 54 dB. One internal wall costs 6 dB. Laptop antenna gain 0 dBi.
4. Received signal: 23 - 54 - 6 = -37 dBm. Very strong.
5. The same at 5 GHz, which loses about 6.4 dB more and where walls cost roughly double: 23 - 72.4 - 24 = -73.4 dBm. Marginal.

Signal (dBm)	Quality	What works
-30 to -50	Excellent	Everything, top rates
-50 to -60	Very good	Video calls, 4K streaming
-60 to -67	Good	Reliable, voice calls fine
-67 to -70	Fair	Streaming may stutter
-70 to -80	Weak	Browsing only, drops likely
below -80	Unusable	Association fails or flaps

6. See it yourself. On macOS: `wdutil info`. On Linux: `iw dev wlan0 link`. On Windows: `netsh wlan show interfaces`.

31.3.4 PLAIN - what is really happening inside

1. Power spreads over the surface of an expanding sphere. Double the distance and the same power covers four times the area, so a fixed antenna catches a quarter as much. That is exactly 6 dB per doubling.
2. Higher frequency loses more for a subtle reason. The spreading is identical, but an antenna sized for a shorter wavelength has a smaller catching area.
3. Multipath means copies arrive at different times. Where two copies arrive half a wavelength apart, they cancel.
4. Half a wavelength at 2.4 GHz is 6 centimetres, which is why moving a laptop by a hand's width can change signal by 10 dB.
5. That cancellation is **fading**, and it is frequency-dependent: one subcarrier may sit in a null while its neighbour is fine. OFDM survives by spreading data across many subcarriers plus error correction.
6. MIMO uses several antennas at each end. Each pair sees a slightly different mix of echoes.
7. With enough difference, the receiver solves a small set of simultaneous equations and separates two or more independent streams sent at the same time on the same channel. Each is a **spatial stream**.
8. **Beamforming** is different. It sends the same data from several antennas with small timing offsets so the copies add at the receiver and cancel elsewhere. It improves signal, not stream count.

31.3.5 TECHNICAL - the engineer's version

1. dBm is decibels relative to one milliwatt: P(dBm) = 10 x log10(P in mW).
2. Free space path loss in decibel form: FSPL(dB) = 20 log10(d_km) + 20 log10(f_MHz) + 32.44.
3. Worked: d = 0.005 km, f = 2437 MHz. 20 log10(0.005) = -46.0. 20 log10(2437) = 67.7. Total = 54.1 dB.
4. Same at 5180 MHz: 20 log10(5180) = 74.3, giving 60.7 dB. The 6.6 dB difference is the 2.4 versus 5 GHz penalty.
5. Thermal noise is N = k x T x B with k = 1.38 x 10^-23 J/K. At 290 K the noise density is -174 dBm/Hz.
6. For 20 MHz: -174 + 10 log10(20,000,000) = -101 dBm. Add a typical 5 dB receiver noise figure and the practical floor is about -96 dBm.

Material	Loss at 2.4 GHz	Loss at 5 GHz
Plasterboard wall	3 to 5 dB	4 to 7 dB
Brick wall	6 to 12 dB	10 to 20 dB
Concrete floor	12 to 20 dB	20 to 30 dB
Clear glass	2 to 4 dB	3 to 6 dB
Human body	3 to 5 dB	5 to 8 dB

7. The honest version of RSSI: in the standard it is a vendor-defined unitless integer with no required scale. Drivers convert it to dBm using their own table, so two adapters in the same spot can legitimately disagree.
8. Tools: `iperf3` for throughput, `iw dev wlan0 station dump` for per-station rate and signal, and Wireshark with a monitor-mode adapter for the radiotap header.

31.3.6 WORDS - remember these

1. Noise floor — the background hiss you must shout over — kTB thermal noise plus receiver noise figure, about -96 dBm in a 20 MHz channel.
2. SNR — the gap between your signal and the hiss — signal-to-noise ratio in decibels, the direct determinant of usable modulation order.
3. Multipath — the same signal arriving by several routes — delayed replicas causing frequency-selective fading.
4. MIMO — several antennas sending different data at once — multiple input multiple output spatial multiplexing.
5. Beamforming — aiming the signal by timing several antennas — applying a steering matrix so transmissions add constructively at the receiver.

31.4 Spectrum and regulation

31.4.1 PLAIN - in simple words

1. Radio is shared. If everyone transmits on the same frequency, nobody gets through, so governments divide the spectrum into bands and say who may use each one, for what, and at what power.
2. Most bands are **licensed**. One company pays for exclusive use. Mobile phone bands work this way.
3. A few bands are **unlicensed**. Anyone may transmit if they obey power limits and interference rules, and may not complain about interference.
4. The main ones are the **ISM bands**, short for industrial, scientific and medical. They were not created for communication. They were set aside for machines that leak radio while doing something else.
5. A microwave oven is the classic example. It cooks with radio at 2.45 GHz. Nobody wanted that band because ovens were already polluting it.
6. The rules differ by country. The rule set in force where you are is called a **regulatory domain**, and your router turns off channels not allowed there.

7. Part of the 5 GHz band is shared with weather and military radar. There, a WiFi device must listen first and leave immediately if it hears a radar. That rule is called DFS.

31.4.2 PLAIN - a picture in your head

1. Think of the spectrum as a very long street of shops. Most of it is rented, one tenant per shop with a locked door. That is licensed spectrum.
2. A few stretches are public squares. Anyone may set up a stall, under rules about how loud you may shout and how big your stall may be.
3. One square sits next to a factory that hums loudly. Rents there are zero. That square is 2.4 GHz.

Where this comparison breaks: a market stall takes a fixed amount of floor. Radio devices share time and frequency together, and a well-behaved device can share a channel with dozens of others by taking short turns. Also the square is public per country, not globally: the same square is open in India and closed in Japan, on the same street.

31.4.3 PLAIN - a worked example

1. The 2.4 GHz band runs from 2400 to 2483.5 MHz in most countries.
2. WiFi defines 14 channels in it, spaced 5 MHz apart, but each channel is about 20 MHz wide, so neighbours overlap heavily.
3. Only channels 1, 6 and 11 are far enough apart not to overlap. That is the origin of the famous advice to use 1, 6 or 11.

Channel	Centre	Overlaps with
1	2412 MHz	2, 3, 4, 5
6	2437 MHz	2 to 5 and 7 to 10
11	2462 MHz	7 to 10, 12, 13
14	2484 MHz	Japan only, 11b only

4. Country differences: the United States allows channels 1 to 11, most of Europe and India allow 1 to 13, and Japan allows 1 to 14 with channel 14 restricted to the old 802.11b modulation.

Sub-band	Range (MHz)	DFS needed
U-NII-1	5150 to 5250	No
U-NII-2A	5250 to 5350	Yes
U-NII-2C	5470 to 5725	Yes
U-NII-3	5725 to 5850	No

5. India is a useful case because it moved recently.
6. The Department of Telecommunications published draft rules in May 2025 and notified the final delicensing on 22 January 2026, opening 5925 to 6425 MHz for indoor low-power and very-low-power wireless access.
7. That is half the band the United States opened in April 2020. India's rules also forbid use on oil platforms, in cars, trains, boats and drones, and on aircraft below 10,000 feet.

31.4.4 PLAIN - what is really happening inside

1. Access points advertise their country in a **Country information element** inside every beacon frame, and a client supporting 802.11d adopts it.
2. DFS works in three steps. Before using a radar channel, the access point listens silently for 60 seconds. That is the **channel availability check**.

3. While using it, the access point keeps listening. On detecting a radar it must stop transmitting within 200 milliseconds and vacate within 10 seconds.
4. It must then avoid that channel for 30 minutes, the **non-occupancy period**.
5. This is why a 5 GHz network sometimes vanishes for a minute and returns on a different channel. It is not a fault. It is the law working.

31.4.5 TECHNICAL - the engineer's version

1. The ISM bands are defined in the ITU Radio Regulations, article 5.
2. The key ones are 902 to 928 MHz (Region 2 only), 2400 to 2500 MHz (global), 5725 to 5875 MHz (global) and 24 to 24.25 GHz.
3. Domestic microwave ovens operate at 2450 MHz plus or minus 50 MHz, which is exactly why that band was designated ISM and why WiFi shares it.
4. The founding regulatory act for WiFi was the United States FCC Report and Order in Docket 81-413, adopted in May 1985, permitting unlicensed spread spectrum in the ISM bands up to 1 watt.
5. Michael Marcus, then an FCC engineer, drove that rulemaking.

Band and region	EIRP limit	Note
2.4 GHz, Europe	20 dBm (100 mW)	Whole band
2.4 GHz, USA	36 dBm point-to-point	30 dBm typical
5150-5250, Europe	23 dBm	Indoor only
6 GHz LPI, India	30 dBm total	Indoor, no outdoor antenna

6. India's regulator for spectrum is the Wireless Planning and Coordination Wing of the Ministry of Communications, written WPC. Equipment approval runs through Equipment Type Approval, ETA.
7. Inspect the applied domain on Linux with `iw reg get`:

```
country IN: DFS-UNSET
    (2402 - 2482 @ 40), (N/A, 20), (N/A)
    (5170 - 5330 @ 160), (N/A, 23), (N/A)
    (5735 - 5835 @ 80), (N/A, 30), (N/A)
```

8. The honest version of "1, 6 and 11 do not overlap": they do not overlap at the -30 dB level used in the spectral mask, but real transmitters leak outside the mask and cheap receivers reject adjacent channels imperfectly.
9. It is a standard that 2.4 GHz channels are 5 MHz apart. It is a convention, not a standard, that everyone uses 1, 6 and 11.

31.4.6 WORDS - remember these

1. ISM band — bands set aside for machines that leak radio — allocations under ITU Radio Regulations article 5.
2. Regulatory domain — the country rule set your radio obeys — a country code mapping to permitted channels, power and DFS requirements.
3. DFS — get off the channel if radar is heard — dynamic frequency selection per ETSI EN 301 893 or FCC Part 15.407.
4. WPC — India's spectrum authority — Wireless Planning and Coordination Wing, Ministry of Communications.

31.5 The WiFi hardware

31.5.1 PLAIN - in simple words

1. A WiFi radio is four things in a row: a digital part, a converter, an analogue radio part, and an antenna.
2. The digital part works out what wave shape should be sent. That is the **baseband** processor.
3. It also runs the rules of taking turns on the air. That part is the **MAC**, short for medium access control.
4. The analogue radio part shifts that wiggle up to 2.4 or 5 GHz, amplifies it, and pushes it to the antenna. Coming in, its most important job is to amplify an extremely weak signal without adding noise.
5. In a modern device the first three are one silicon chip. Around it sit a power amplifier, a low-noise amplifier, switches and filters, together called the **RF front end**.
6. The chip runs its own small program, stored as **firmware**. Your operating system does not do the split-second radio timing. The chip does.

31.5.2 PLAIN - a picture in your head

1. Think of a small radio station with four rooms.
2. A producer writes the script and decides when the station goes on air. That is the MAC and the baseband.
3. A machine turns the script into sound, and a transmitter lifts that sound onto the broadcast frequency where an amplifier makes it loud. That is the converter, the mixer and the power amplifier.
4. Incoming signals run the same rooms backwards, and the most delicate part is the first receive stage, because whatever noise it adds is amplified along with the tiny signal forever after. That stage is the low-noise amplifier.

Where this comparison breaks: a radio station transmits or receives. A WiFi chip must do both, alternating thousands of times per second, sharing one antenna through a switch that must settle in under a few microseconds.

31.5.3 PLAIN - a worked example

1. Open a laptop from around 2020 and you find a small card in an M.2 2230 slot, about 22 mm by 30 mm.
2. Two thin coaxial wires run from that card up through the hinge into the lid, ending at flat metal shapes printed just above the screen.
3. Those are the antennas, placed high and away from the metal chassis and your hands.

Part	Job	Typical figure
MAC and baseband	Frames, timing, OFDM	2x2 spatial streams
RF transceiver	Up and down conversion	2.4, 5 and 6 GHz
Power amplifier	Boost transmit signal	+18 to +20 dBm
Low-noise amplifier	Boost received signal	2 to 3 dB noise figure
Antenna switch	Share antenna Tx/Rx	Under 1 us switching

4. In a phone there is no card. WiFi, Bluetooth and often GPS sit in one **combo** chip soldered to the main board.
5. Phone antennas are usually part of the metal frame, split into segments by the thin plastic gaps you can see on the outside edge.
6. Real examples: Apple used Broadcom BCM4387-family parts in iPhone 13 and 14 era devices, and Qualcomm's FastConnect 7800, announced in 2022, was the first widely shipped WiFi 7 combo.

31.5.4 PLAIN - what is really happening inside

1. Follow one packet out of the chip. The MAC has a frame of bytes and permission to transmit.
2. The baseband adds error correction, maps groups of bits onto constellation points, and lays those points across the OFDM subcarriers.

3. An inverse fast Fourier transform turns that set of subcarriers into a stream of numbers describing the wave shape over time.
4. Converters turn those numbers into two voltages, I and Q, and a mixer multiplies them by a pure tone from a **local oscillator** to shift everything up to 2.4 or 5 GHz.
5. The power amplifier raises it from a few milliwatts to 60 or 100 milliwatts, a switch connects the antenna, and the wave leaves.
6. Coming in, the low-noise amplifier boosts the tiny signal first, before anything can add noise.
7. A mixer shifts it down, converters digitize it, an FFT splits it into subcarriers, an equalizer undoes the channel distortion using known training symbols at the start of the frame, and a decoder recovers bits.
8. The firmware handles anything faster than the operating system can be trusted to react: acknowledgements within 16 microseconds, backoff countdown, rate selection and power save timing.

31.5.5 TECHNICAL - the engineer's version

Interface	Typical use	Note
PCIe	Laptop M.2 cards	Highest throughput
SDIO	Embedded, older phones	Limited to ~200 Mbit/s
USB	Dongles	Latency and jitter issues
CNVi	Intel laptops since 2018	MAC moved into the chipset

1. CNVi is an instructive implementation detail: Intel moved the MAC and baseband into the platform controller hub, leaving only RF on the M.2 card. This is why Intel CNVi cards do not work in AMD laptops.
2. Firmware is loaded at driver initialization from files on the host. On Linux they live under `/lib/firmware`:

```
/lib/firmware/iwlwifi-ty-a0-gf-a0-89.ucode      Intel AX210
/lib/firmware/brcm/brcmfmac43455-sdio.bin       Broadcom in Pi 4
/lib/firmware/ath11k/QCA6390/hw2.0/amss.bin     Qualcomm
/lib/firmware/mediatek/WIFI_RAM_CODE_MT7922_1.bin
```

3. Check what is loaded with `dmesg | grep -i firmware` or `ethtool -i wlan0`, which prints driver, firmware version and bus address.
4. Nearly all WiFi firmware is a closed binary. That is a real limitation for auditability, and it has hosted serious vulnerabilities, notably Google Project Zero's 2017 findings against Broadcom WiFi firmware.
5. Two design styles exist. **FullMAC** puts scanning, association and much of the MAC in firmware. **SoftMAC** puts them in the host driver.
6. Most Broadcom and Qualcomm mobile parts are FullMAC; most Atheros and Intel parts on Linux are SoftMAC, which is why they support monitor mode better.

31.5.6 WORDS - remember these

1. Baseband — the part that shapes the wave from bits — PHY layer coding, mapping, IFFT and FFT.
2. MAC — the part that decides when to talk — medium access control, the 802.11 state machine and timing.
3. Low-noise amplifier — the delicate first receive stage — LNA whose noise figure sets the receiver's sensitivity.
4. Firmware — the program running on the radio chip itself — closed binary loaded from the host, handling microsecond timing.
5. SoftMAC — the computer handles connecting — MAC state machine in the host driver, as with Linux mac80211.

31.6 WiFi history and standards

31.6.1 PLAIN - in simple words

1. WiFi did not start as a product idea. It started as a regulator giving away spectrum nobody wanted.
2. In 1985 the United States regulator allowed unlicensed devices to use the ISM bands if they used spread spectrum. That made it legal to build a wireless network without buying spectrum.
3. The way devices take turns was borrowed from ALOHAnet, built at the University of Hawaii and operating from June 1971.
4. ALOHAnet connected islands by radio at 9600 bits per second. Its rule was simple: transmit when you have something to send, and if no acknowledgement comes, wait a random time and try again.
5. That idea, random access with random backoff, became Ethernet and then WiFi.
6. The first WiFi standard, IEEE 802.11, arrived in 1997 at 1 or 2 megabits per second. It was too slow to matter.
7. In 1999 two versions appeared: 802.11b at 11 megabits in 2.4 GHz, and 802.11a at 54 megabits in 5 GHz. 802.11b won, because 2.4 GHz radios were cheaper and the range was better.
8. The name WiFi was invented in 1999 by a branding company. It does not stand for anything.

31.6.2 PLAIN - a picture in your head

1. Think of WiFi generations as widening a road and adding lanes.
2. 802.11b was one narrow lane. 802.11a and g repaved it with a better surface, OFDM instead of DSSS, for five times the traffic.
3. 802.11n added lanes, with several antennas sending different data at once, and 802.11ac widened each lane from 20 MHz to 80 and then 160.
4. 802.11ax neither widened nor added lanes. It fixed the junction, letting several small vehicles cross together instead of one at a time.
5. 802.11be opened a second road, the 6 GHz band, and lets a device use two roads at once.

Where this comparison breaks: roads do not interfere with each other. Radio channels do, and adding width in a crowded area can make things worse. A 160 MHz channel in a block of flats often performs worse than an 80 MHz one.

31.6.3 PLAIN - a worked example

1. Here is the complete family. Read the rate column as a theoretical ceiling no real device reaches.

Standard	Name and year	Band	Max PHY rate
802.11-1997	none, 1997	2.4 GHz	2 Mbit/s
802.11b	WiFi 1, 1999	2.4 GHz	11 Mbit/s
802.11a	none, 1999	5 GHz	54 Mbit/s
802.11g	WiFi 3, 2003	2.4 GHz	54 Mbit/s
802.11n	WiFi 4, 2009	2.4 and 5	600 Mbit/s
802.11ac	WiFi 5, 2013	5 GHz	6.93 Gbit/s
802.11ax	WiFi 6, 2019/21	2.4/5/6	9.6 Gbit/s
802.11be	WiFi 7, 2024	2.4/5/6	about 46 Gbit/s

Standard	Key feature added
802.11b	DSSS and CCK, first mass adoption
802.11a	OFDM, 5 GHz band
802.11g	OFDM at 2.4 GHz, b compatible

Standard	Key feature added
802.11n	MIMO, 40 MHz, frame aggregation
802.11ac	80/160 MHz, 256-QAM, MU-MIMO down
802.11ax	OFDMA, 1024-QAM, TWT, BSS colour
802.11be	320 MHz, 4096-QAM, multi-link

2. Note the numbering gap. The Wi-Fi Alliance introduced generation numbers in October 2018 and applied them backwards only to n, ac and ax. "WiFi 1" through "WiFi 3" for b, a and g are retrospective informal labels.
3. WiFi 6E is not a new standard. It is 802.11ax operating in the 6 GHz band, branded separately from 2021.

31.6.4 PLAIN - what is really happening inside

1. Now the Australian claim, told honestly, because it is often told badly.
2. In the early 1990s a team at CSIRO, Australia's national science agency, was working on a radio astronomy problem: detecting a weak signal smeared by echoes.
3. The team was led by John O'Sullivan, with Graham Daniels, Terence Percival, Diethelm Ostry and John Deane.
4. Their solution combined forward error correction, frequency-domain interleaving and multi-carrier modulation, done with fast Fourier transforms in a single chip.
5. That combination was granted as US Patent 5,487,069 on 23 January 1996.
6. CSIRO settled with Hewlett-Packard in April 2009, then Microsoft, Intel, Dell, Toshiba, Asus, Nintendo and others in 2009 and 2010, and with AT&T, Verizon and T-Mobile in 2012 for about 229 million dollars.
7. Total income from the patent was roughly 430 million dollars, and the inventors received the European Inventor Award in 2012.
8. What is not true is that CSIRO invented WiFi.
9. OFDM itself was described by Chang at Bell Labs in 1966, and the 802.11 committee was chaired by Vic Hayes of NCR in the Netherlands, whose team built the first products.
10. The honest version: WiFi has no single inventor.
11. The FCC opened the band, NCR and Lucent built the first radios, the IEEE committee wrote the rules, CSIRO contributed a key patented signal processing idea, and Apple shipping AirPort in the 1999 iBook made it mainstream.

31.6.5 TECHNICAL - the engineer's version

1. IEEE 802.11 formed as a working group in 1990. Vic Hayes of NCR chaired it for ten years and is commonly called the father of WiFi.
2. The 1997 base standard defined three physical layers: FHSS at 2.4 GHz, DSSS at 2.4 GHz, and diffuse infrared at 1 and 2 Mbit/s. The infrared PHY was never commercially implemented.
3. 802.11b, ratified September 1999, added CCK for 5.5 and 11 Mbit/s. 802.11a, ratified the same month, used OFDM at 5 GHz with rates 6 to 54 Mbit/s.
4. The Wireless Ethernet Compatibility Alliance was founded in 1999 and renamed the Wi-Fi Alliance in 2002. Interbrand coined the name WiFi in 1999 from a list of ten candidates.
5. The "Wireless Fidelity" expansion is a myth. Phil Belanger, a founding member, has stated the name has no expanded meaning; early Alliance marketing briefly used the phrase and created the confusion.
6. 802.11ax was published as IEEE 802.11ax-2021, with Wi-Fi Alliance certification from September 2019. It introduced OFDMA, uplink MU-MIMO, 1024-QAM, target wake time and BSS colouring.
7. 802.11be certification as Wi-Fi 7 began on 8 January 2024, with the IEEE standard published 22 July 2025. Key additions: 320 MHz channels, 4096-QAM, multi-link operation, multiple resource units and preamble puncturing.

8. Rate arithmetic for 802.11be: 8 spatial streams, 320 MHz, 4096-QAM, 5/6 coding and a 0.8 us guard gives about 23 Gbit/s. The headline 46 Gbit/s assumes the specification maximum of 16 spatial streams, which no product implements.
9. Wi-Fi 8 is IEEE 802.11bn, under the banner Ultra High Reliability, projected to be finalized around May 2028.
10. Its stated goal is not higher peak rate but better worst-case behaviour: lower latency variance, seamless roaming and coordinated access points. Treat published figures as active work, not established fact.

31.6.6 WORDS - remember these

1. IEEE 802.11 — the committee and document that defines WiFi — the working group and its standard, amended by letter suffixes.
2. Wi-Fi Alliance — the industry group that tests and brands — the certifier owning the WiFi trademark and generation numbers.
3. 802.11ax — the standard behind WiFi 6 and 6E — high efficiency WLAN with OFDMA and 1024-QAM, published 2021.
4. 802.11be — the standard behind WiFi 7 — extremely high throughput with 320 MHz, 4096-QAM and multi-link operation, published July 2025.
5. CSIRO patent — the Australian signal processing patent WiFi uses — US 5,487,069 of January 1996, covering FEC with interleaved multicarrier modulation.

31.7 How WiFi actually works on the air

31.7.1 PLAIN - in simple words

1. An access point announces itself about ten times a second with a short message called a **beacon**, carrying the network name, supported speeds, security type and a timestamp.
2. Your device can sit quietly and listen for beacons, called **passive scanning**, or shout a question and wait for answers, called **active scanning**. The shout is a probe request.
3. The network name you see is the **SSID**. The unique hardware address of the access point radio is the **BSSID**.
4. To join, your device sends an authentication frame, then an association request. The access point replies and gives it a small ID number. Then the security handshake runs, and only then can real data flow.
5. The rule is: listen first. If the channel is busy, wait. If it is free, wait a random extra amount, then transmit.
6. Every unicast frame must be acknowledged. No acknowledgement means it was lost and is sent again.
7. All that listening, waiting and acknowledging costs time. Roughly half the advertised speed disappears into it.

31.7.2 PLAIN - a picture in your head

1. Think of a polite conference call with no chairperson.
2. Nobody can hear themselves while speaking, because their own voice drowns everything out, so nobody can tell if they talked over someone.
3. That is exactly the radio situation: a transmitter cannot listen on its own channel while transmitting, because its own signal is a billion times stronger than anything arriving.
4. The etiquette is: wait for silence, count silently for a random number, and if it is still silent when you reach zero, start talking.
5. Now a problem: two people at opposite ends of a long table can both hear the chairperson but not each other, and will talk over each other forever.
6. The fix is to ask the chairperson “may I speak”, and have the chairperson say out loud “yes, everyone else be quiet for two minutes”.

Where this comparison breaks: humans hear a clash afterwards and apologize. Radios never learn about the clash directly, only from the missing acknowledgement. Also the "random count" is not seconds, it is a count of 9 microsecond slots, and the range doubles after every failure.

31.7.3 PLAIN - a worked example

1. Here is a full join sequence, in order.

```
Client                                Access point
  |                                        |
  |<---------- Beacon (every 102.4 ms) ---|
  |------ Probe Request (broadcast) ----->|
  |<----- Probe Response ------------------|
  |------ Authentication (open) --------->|
  |<----- Authentication (success) -------|
  |------ Association Request ----------->|
  |<----- Association Response (AID 3) ---|
  |<----- EAPOL 4-way handshake msg 1 ----|
  |------ EAPOL msg 2 ------------------->|
  |<----- EAPOL msg 3 --------------------|
  |------ EAPOL msg 4 ------------------->|
  |         (keys installed, port open)   |
  |------ DHCP Discover ----------------->|
  |<----- DHCP Offer / Ack --------------|
  |         data flows                    |
```

2. Typical times: probe response within 10 milliseconds, authentication and association a few milliseconds, four-way handshake 20 to 60 milliseconds, DHCP 10 to 100 milliseconds. Total join time is usually 100 to 300 milliseconds.
3. Now the taking-turns arithmetic for one 1500 byte packet on a 5 GHz 802.11ac link advertising 866 Mbit/s.

Step	Time
DIFS wait	34 us
Average random backoff	67 us
Preamble and header	20 us
1500 bytes at 866 Mbit/s	14 us
SIFS gap	16 us
Acknowledgement frame	24 us
Total	175 us

4. Useful data moved: 12,000 bits in 175 microseconds is 68.5 Mbit/s, against an advertised 866.
5. That is why frame aggregation exists. Send 64 frames in one burst with one preamble and one block acknowledgement, and the fixed costs are shared.
6. With aggregation the same link delivers roughly 500 to 600 Mbit/s of real throughput. Still not 866, but far better than 68.

31.7.4 PLAIN - what is really happening inside

1. **SIFS**, short interframe space, is the shortest gap, used for replies that must not be interrupted, such as an acknowledgement.
2. **DIFS**, distributed interframe space, is longer, and is what a device must see idle before starting a fresh transmission.
3. After DIFS, the device picks a random number between 0 and the contention window and counts down one slot at a time.

4. If the channel goes busy mid-count, the count freezes and resumes after the next idle DIFS. It does not restart, which gives waiting devices fairness.
5. On failure the contention window doubles: 15, 31, 63, 127, up to 1023. That is **exponential backoff**, the same idea Ethernet uses.
6. The hidden node problem is common: two clients on opposite sides of an access point may be unable to hear each other.
7. The fix is **RTS/CTS**. The sender asks with a request to send, and the access point answers with a clear to send that every client hears.
8. The clear to send carries a duration. Everyone who hears it sets a countdown called the **network allocation vector** and stays silent for that long, even without hearing the channel busy. That is **virtual carrier sense**.
9. Every 802.11 frame starts with a 2 byte frame control field saying what kind it is, then a duration, then up to four addresses.

31.7.5 TECHNICAL - the engineer's version

Parameter	2.4 GHz DSSS	5 GHz OFDM
Slot time	20 us	9 us
SIFS	10 us	16 us
DIFS	50 us	34 us
CWmin	31	15
CWmax	1023	1023

1. The 802.11 MAC header layout, in bytes:

```
+----+----+------+------+------+-----+------+-----+------+
| FC | Dur| Addr1| Addr2| Addr3| Seq | Addr4| QoS | Body |
|  2 |  2 |   6  |   6  |   6  |  2  |  6*  |  2* |  ... |
+----+----+------+------+------+-----+------+-----+------+
                                          then FCS: 4

* present only in some frame types
```

2. Aggregation: A-MSDU packs several IP packets under one MAC header, up to 11,454 bytes in 802.11n. A-MPDU packs several MAC frames under one PHY preamble, up to 1 megabyte in 802.11ac and ax, with a block ACK.
3. As a rule of thumb, TCP throughput lands at 45 to 60 percent of the PHY rate for 802.11n and ac, and 60 to 70 percent for 802.11ax with OFDMA, because OFDMA removes much of the per-station contention.
4. **OFDMA** in 802.11ax divides the channel into resource units measured in subcarriers, called tones.

RU size	Tones	Users in 20 MHz
26-tone	26	up to 9
52-tone	52	up to 4
106-tone	106	up to 2
242-tone	242	1 (whole channel)

5. The access point schedules who uses which resource unit with a trigger frame, so uplink transmissions from several clients arrive simultaneously and must be time-aligned within 0.4 microseconds.
6. **Target wake time** lets a client and access point agree a future window for traffic so the client sleeps in between. It is the single biggest battery feature in 802.11ax and matters most for sensors.
7. Band steering is a vendor behaviour, not a standard: the access point withholds or delays probe responses on 2.4 GHz to push dual-band clients to 5 GHz.

8. The related real standards are 802.11k for neighbour reports, 802.11v for BSS transition management and 802.11r for fast roaming.
9. Capture and inspect: `iw dev wlan0 set type monitor`, then `tcpdump -i wlan0 -e -s0` or Wireshark with the display filter `wlan.fc.type_subtype == 0x08` for beacons.

31.7.6 WORDS - remember these

1. Beacon — the regular announcement an access point sends — a management frame every 100 TU carrying SSID, capabilities and timestamp.
2. BSSID — the identity of one specific access point radio — the MAC address of that radio, unique per band per access point.
3. CSMA/CA — listen, wait a random time, then talk — carrier sense multiple access with collision avoidance and exponential backoff.
4. SIFS — the tiny gap before a reply — short interframe space, 16 us on 5 GHz OFDM, giving replies priority over new transmissions.
5. RTS/CTS — asking permission out loud so hidden devices hear — request to send and clear to send, setting the network allocation vector.
6. OFDMA — several devices sharing one transmission by frequency — scheduled resource units within a channel.
7. TWT — an agreed appointment so a device can sleep — target wake time, introduced in 802.11ax.

31.8 WiFi security

31.8.1 PLAIN - in simple words

1. On radio, everything is in the air and anyone in range receives every bit, so WiFi must encrypt by default and must prove who is who.
2. The first attempt, **WEP** from 1997, was broken beyond repair within four years.
3. **WPA** in 2003 was a patch that ran on the same old hardware. It bought time.
4. **WPA2** in 2004 was the real fix, using the AES cipher. It is still everywhere.
5. **WPA3**, announced in 2018, fixed the last big weakness: an attacker who captured your handshake could guess your password offline, forever.
6. There is also open, with no password, where every byte is readable by anyone in range unless the site itself uses HTTPS.

31.8.2 PLAIN - a picture in your head

1. Think of a locked diary with a code wheel on the front.
2. WEP used one short code and wrote part of it on the outside of every page. After watching enough pages, you could work out the whole code.
3. WPA2 uses a proper safe. The password is not the key: the password plus a fresh random number is stirred into a key used only for this session.
4. WPA3 changes the opening ceremony so the recording is useless. Each guess must be made live, against the real router, one at a time.

Where this comparison breaks: a diary code is one secret. WiFi has a hierarchy of keys, one master key from the password, one pairwise key per client, one group key for broadcasts. Also, encryption stops at the access point. Everything past it is in the clear unless the application encrypts too.

31.8.3 PLAIN - a worked example

1. The absolute rule of a stream cipher is that a keystream must never be used twice. If it is, adding the two ciphertexts cancels the keystream.
2. To vary the keystream, WEP prepends a 24-bit **initialization vector** to the shared key and sends that vector in the clear with every packet.

3. 24 bits gives 16,777,216 values. By the birthday problem a repeat becomes likely after about 5,000 packets, which a busy access point sends in seconds.
4. Worse, RC4 has a related-key weakness, published as the FMS attack by Scott Fluhrer, Itsik Mantin and Adi Shamir in August 2001.
5. The 2007 PTW attack by Erik Tews, Andrei Pyshkin and Ralf-Philipp Weinmann recovers a 104-bit WEP key from about 40,000 packets in under a minute.
6. Now the WPA2 four-way handshake. Both sides already know a master key derived from the password and the network name.

```
AP                                          Client
 |--- 1. ANonce (AP random number) --------->|
 |     client now has both nonces and both   |
 |     MACs, so it computes the PTK          |
 |<-- 2. SNonce + MIC (proves it has PTK) ---|
 |     AP computes the same PTK and checks   |
 |--- 3. GTK (encrypted) + MIC ------------->|
 |<-- 4. Acknowledgement + MIC --------------|
 |     both install keys, data port opens    |
```

7. Nothing secret crosses the air. Both sides derive the same session key from things they already share plus two random numbers.

31.8.4 PLAIN - what is really happening inside

1. The password and the SSID are stretched into a 256-bit **pairwise master key** by 4096 rounds of hashing. That slowness is deliberate, to make guessing expensive.
2. The four-way handshake mixes that master key with two fresh random numbers and both MAC addresses to make a **pairwise transient key** for this session only.
3. The KRACK attack of October 2017, found by Mathy Vanhoef and Frank Piessens at KU Leuven in Belgium, attacked this handshake.
4. It blocks message four so the access point resends message three. The client reinstalls the same key and resets its packet counter.
5. Reusing a counter with a stream-like cipher exposes the keystream, which is exactly the WEP failure in a new place.
6. On Android and Linux the wpa_supplicant implementation was worse: it installed an all-zero key on reinstallation, removing encryption entirely. That was fixed in wpa_supplicant 2.7.
7. WPA3 replaces the password-to-key step with **SAE**, simultaneous authentication of equals, a password-authenticated key exchange.
8. With SAE a captured handshake cannot be used for offline guessing, and each session gets forward secrecy: learning the password later does not decrypt old recorded traffic.

31.8.5 TECHNICAL - the engineer's version

Scheme	Year	Cipher	Status
WEP	1997	RC4, 24-bit IV	Broken, deprecated 2004
WPA	2003	RC4 with TKIP	Deprecated 2012
WPA2	2004	AES-CCMP	Still standard
WPA3	2018	AES-CCMP, SAE	Mandatory for new certs

1. WPA2 is IEEE 802.11i-2004. CCMP is AES in counter mode with CBC-MAC, 128-bit key, 48-bit packet number and an 8-byte message integrity code.
2. TKIP was a stopgap that kept RC4 but added per-packet key mixing, a 48-bit sequence counter and the Michael integrity check. The Wi-Fi Alliance barred TKIP-only devices from certification in 2012.

3. Key derivation is PMK = PBKDF2-HMAC-SHA1(passphrase, SSID, 4096 iterations, 256 bits). Because the SSID is the salt, common names such as "linksys" have precomputed rainbow tables.
4. KRACK carries ten CVEs, CVE-2017-13077 through CVE-2017-13088.
5. **OWE**, opportunistic wireless encryption, branded Enhanced Open, encrypts open networks with an unauthenticated Diffie-Hellman exchange. It stops passive sniffing but not an active impostor access point.
6. **WPS** is dangerous. Its 8-digit PIN is validated in two halves and the last digit is a checksum, so the search space is 11,000, not 100 million.
7. Stefan Viehbock published this in December 2011, and Dominique Bongard's 2014 Pixie Dust attack breaks many chipsets offline in seconds. Turn it off.
8. Enterprise uses IEEE 802.1X: the client is the supplicant, the access point the authenticator, and a RADIUS server the authentication server.
9. EAP carries the method, with EAP-TLS using client certificates and PEAP-MSCHAPv2 or EAP-TTLS tunnelling a password inside TLS.
10. Tools: `wpa_cli status`, `hostapd_cli`, `aircrack-ng` for auditing your own network, and Wireshark with the display filter `eapol` to watch the four-way handshake.

31.8.6 WORDS - remember these

1. WEP — the broken first attempt — RC4 with a 24-bit IV, key recoverable from 40,000 packets by the PTW attack.
2. WPA2 — the long-standing standard — IEEE 802.11i with AES-CCMP and the four-way handshake.
3. Four-way handshake — the exchange proving both sides know the password — EAPOL-Key frames deriving the PTK from PMK, two nonces and two MACs.
4. KRACK — the 2017 attack that replayed a handshake message — key reinstallation causing nonce reuse.
5. WPA3 — the current standard — SAE replacing the PSK exchange, giving forward secrecy and no offline dictionary attack.
6. WPS — the push-button feature you should disable — Wi-Fi Protected Setup, whose PIN has an effective search space of 11,000.

31.9 The WiFi software stack

31.9.1 PLAIN - in simple words

1. Five layers of software sit between your click and the air.
2. **Firmware** runs on the WiFi chip itself and handles anything measured in microseconds.
3. The **driver** runs inside the operating system kernel and talks to the chip over PCIe or SDIO.
4. The **802.11 stack** implements the rules of scanning, joining and framing for drivers that do not do it in firmware.
5. The **supplicant** is a background program that does the security handshake and holds your passwords.
6. The **network manager** is the part with the menu you click. It decides which network to join and asks the supplicant to do it.
7. Your password is not sent to the router. It is used locally to compute a key. This is worth repeating because almost everyone assumes otherwise.

31.9.2 PLAIN - a picture in your head

1. Think of joining a members-only club.
2. The network manager is you, reading the list of clubs and picking one.
3. The supplicant is your solicitor, who holds your documents and handles the paperwork at the door.
4. The 802.11 stack is the club's standard entry procedure: queue here, show this, sign that.
5. The driver is the doorman who opens the door, and the firmware is the door mechanism, which must latch in microseconds and cannot wait for anyone.
6. Once inside, DHCP is the receptionist who assigns you a table number.

Where this comparison breaks: in a club these are separate people. On a FullMAC phone chip the procedure, the doorman and the mechanism are one closed piece of firmware, and the operating system only sends high-level requests such as "join this network".

31.9.3 PLAIN - a worked example

1. Here is exactly what happens in software when you click a network name and type a password. This is the Linux path, the one you can read.

```
 1. You click. The GUI calls NetworkManager over D-Bus.
 2. NetworkManager writes a connection profile file.
 3. NetworkManager tells wpa_supplicant over D-Bus:
    "network={ssid=..., psk=...}, select it".
 4. wpa_supplicant asks the kernel to scan, via nl80211.
 5. cfg80211 passes the request to the driver; the driver
    programs the chip to sweep channels.
 6. Scan results come back up the same path as a list of
    BSSIDs with signal, channel and security info.
 7. wpa_supplicant picks the best BSSID for that SSID.
 8. It computes PMK = PBKDF2(password, SSID, 4096, 256).
 9. It asks the kernel to authenticate and associate.
10. mac80211 sends the auth and assoc frames on the air.
11. The AP replies; the association succeeds.
12. The AP sends EAPOL message 1. The driver passes it up.
13. wpa_supplicant computes the PTK and sends message 2.
14. Messages 3 and 4 complete. Keys go to the kernel.
15. The kernel installs keys into the chip's key table.
16. The netdev is marked carrier-up. NetworkManager sees it.
17. NetworkManager runs its DHCP client. An address arrives.
18. Routes and DNS servers are installed. You are online.
```

2. Steps 1 to 3 take milliseconds. The scan is the slow part: a full sweep of all channels takes 1 to 4 seconds, because it must dwell on each channel long enough to catch a beacon.
3. Steps 8 to 15 take 20 to 100 milliseconds, and step 17 takes 10 to 100.

Platform	Where the password is stored
Linux, NetworkManager	/etc/NetworkManager/
Linux, plain supplicant	/etc/wpa_supplicant.conf
macOS	login and System keychains
Windows	DPAPI-protected profile XML
Android	/data/misc/wifi, keystore-backed

4. A minimal supplicant configuration looks like this:

```
ctrl_interface=/run/wpa_supplicant
update_config=1
```

```
network={
    ssid="HomeNet"
    psk="a-long-passphrase"
    key_mgmt=WPA-PSK
    priority=10
}
```

31.9.4 PLAIN - what is really happening inside

1. Reconnecting later skips most of that, because the profile is already stored: no typing and no dialogue.
2. The device keeps a list of known networks and scans for them on waking.
3. Historically it also sent probe requests naming each known SSID, which leaked your home and office network names to anyone listening. Modern devices prefer passive scanning and randomized MAC addresses.
4. If the master key has not expired, **PMK caching** lets a device reassociate to a previously used access point without the expensive derivation. Standards 802.11r and opportunistic key caching extend this between access points in one network.
5. Roaming decisions live in the client, not the network. The client watches signal and loss, and jumps when a better BSSID crosses a threshold.
6. That is why a laptop sometimes clings to a distant access point: the thresholds are a vendor implementation detail, not a standard, and vary widely between chipsets.
7. On macOS the same pieces exist with different names: the driver family is IO80211Family, network selection lives in `airportd`, applications use the CoreWLAN framework, and passwords sit in the keychain as items of kind “AirPort network password”.

31.9.5 TECHNICAL - the engineer’s version

1. The Linux layering, top to bottom:

```
GUI  ->  NetworkManager or iwd   (userspace policy)
           |  D-Bus
         wpa_supplicant          (802.1X, WPA, SAE)
           |  nl80211 over netlink
         cfg80211                (config API, regulatory)
           |
         mac80211                (SoftMAC 802.11 state machine)
           |
         driver: iwlwifi / ath11k / mt76 / brcmfmac
           |  PCIe, SDIO, USB
         firmware on the chip
```

2. cfg80211 is the configuration layer and holds the regulatory database. mac80211 is the SoftMAC implementation, used only by drivers that need it; FullMAC drivers such as brcmfmac register directly with cfg80211.

Command	Shows
`iw dev wlan0 link`	BSSID, signal, current bitrate
`iw dev wlan0 scan`	Beacon contents of neighbours
`iw reg get`	Active regulatory domain
`wpa_cli status`	Supplicant state, key management
`nmcli device wifi list`	NetworkManager’s view
`ethtool -i wlan0`	Driver and firmware versions

3. On macOS 13 and later, `sudo wdutil info` prints RSSI, noise, channel, security, country code and the MAC address in use. The older `airport` utility inside Apple80211.framework was removed in macOS 14.4.
4. The security boundary is worth stating plainly: your operating system trusts the WiFi firmware completely, and on PCIe that firmware has DMA access unless an IOMMU is configured. That is an established weakness, not a theoretical one.
5. Debugging: `dmesg -w` while connecting shows driver and firmware messages, and `wpa_supplicant -dd -i wlan0 -c file.conf` in the foreground prints every state transition, which is the fastest way to see where a join fails.

31.9.6 WORDS - remember these

1. Supplicant — the program that does the security handshake — 802.1X supplicant, typically wpa_supplicant or iwd on Linux.
2. mac80211 — the kernel's 802.11 rules engine — the SoftMAC framework implementing scanning, association and rate control.
3. Network manager — the part with the menu — the userspace policy daemon choosing networks and running DHCP.
4. PMK caching — remembering a key so reconnecting is fast — caching the pairwise master key security association per BSSID.

31.10 Bluetooth

31.10.1 PLAIN - in simple words

1. Bluetooth solves a different problem to WiFi. WiFi connects you to a network. Bluetooth connects two devices to each other.
2. It uses the same 2.4 GHz band as WiFi but very differently: it hops between narrow channels 1600 times a second.
3. A small group of connected devices is a **piconet**. One device is in charge of timing and the others follow.
4. Because the purposes vary so much, Bluetooth defines **profiles**: agreed rules for one job, such as stereo audio, a keyboard, or a phone call.
5. **Pairing** is the one-time ceremony creating a shared secret, so the two devices recognize each other later without asking again.

31.10.2 PLAIN - a picture in your head

1. Think of a busy hall where many pairs of people hold private conversations.
2. Each pair agrees a rule: after every sentence, move to a different spot, following a shuffle order only they know.
3. Pairing is the moment two people first exchange a private codeword, in person, so they can recognize each other later in the dark.

Where this comparison breaks: people would collide constantly while moving. The hopping pattern is derived from the leader's address and clock, so every follower knows exactly where the leader will be next. Also, since Bluetooth 1.2 in 2003, devices drop persistently busy channels from the hop list, which no wandering person could do.

31.10.3 PLAIN - a worked example

Version	Year	What it added
1.0	1999	First release, 723 kbit/s

Version	Year	What it added
1.2	2003	Adaptive hopping, faster connect
2.0 + EDR	2004	Enhanced data rate, 2.1 Mbit/s
2.1 + EDR	2007	Secure Simple Pairing
3.0 + HS	2009	WiFi used for bulk transfer
4.0	2010	Bluetooth Low Energy, GATT
4.2	2014	Better privacy, IPv6 over BLE
5.0	2016	BLE 2 Mbit/s or 4x range
5.1	2019	Direction finding, AoA and AoD
5.2	2019	LE Audio, LC3 codec, isochronous
5.3	2021	Connection subrating, efficiency
5.4	2023	Encrypted advertising, PAwR
6.0	2024	Channel sounding for distance

1. The radio numbers: 79 channels of 1 MHz each from 2402 to 2480 MHz, hopped 1600 times a second, so each hop lasts 625 microseconds.

Profile	Job
A2DP	Stereo music to a speaker
AVRCP	Play, pause, skip, volume
HFP	Phone calls, two-way voice
HID	Keyboard, mouse, game pad
SPP	A plain serial cable replacement
PBAP	Sharing contacts with a car

31.10.4 PLAIN - what is really happening inside

1. One device in a piconet is the **central**, historically called master, and up to seven others are actively connected **peripherals**.
2. The central's clock and address generate the hop sequence. Time is divided into 625 microsecond slots and the central decides who transmits in each.
3. Above the radio sit a link layer, a multiplexing layer called L2CAP, then the profiles. Profiles exist so a headset from one maker works with a phone from another with no shared code.
4. **Legacy pairing**, before 2007, used a short PIN, often 0000, and a passive listener who captured the exchange could recover the link key.
5. **Secure Simple Pairing**, in Bluetooth 2.1 from 2007, uses elliptic curve Diffie-Hellman so a passive listener learns nothing.
6. It offers four association models, chosen by what each device can display and what you can type into it.
7. **Numeric comparison**: both show a six-digit number and you confirm they match. This is the strongest common model.
8. **Passkey entry**: one shows a number and you type it into the other, used for keyboards.
9. **Just Works**: neither device can display or input, so nothing is checked. It stops passive eavesdropping but not an active impostor.
10. The result is a **link key** stored on both sides. Storing it is **bonding**, and it is why a headset reconnects silently forever after.

31.10.5 TECHNICAL - the engineer's version

1. Development began at Ericsson Mobile in Lund, Sweden, where Nils Rydbeck started a short-link radio project in 1989. Design work began in 1994, led by Jaap Haartsen with Sven Mattisson.
2. The Bluetooth Special Interest Group launched in May 1998 with five founding members: Ericsson, IBM, Intel, Nokia and Toshiba. The specification was announced on 20 May 1998.
3. The name was proposed by Jim Kardach of Intel in 1997 after Harald "Bluetooth" Gormsson, the tenth-century Danish king who united the Danish tribes. It was meant as a code name and stuck.
4. The controller and host are separated by the **HCI**, host controller interface, over UART, USB or SDIO. That split is why one chip works with Linux BlueZ, Windows and Android alike.
5. Attacks worth naming: BlueBorne, published by Armis in September 2017, gave remote code execution with no pairing.
6. KNOB, by Antonioli, Tippenhauer and Rasmussen in 2019, forced the encryption key down to one byte of entropy. BIAS in 2020 impersonated a previously bonded device.
7. The honest version of "Bluetooth is secure": the cryptography in Secure Connections mode, added in 4.2 using P-256 ECDH and AES-CCM, is sound.
8. The weaknesses are almost all in negotiation, downgrade paths and Just Works, plus a long tail of unpatched embedded devices.

31.10.6 WORDS - remember these

1. Piconet — a small group sharing timing — one central and up to seven active peripherals on a common hop sequence.
2. Frequency hopping — jumping between narrow channels constantly — 1600 hops per second across 79 channels of 1 MHz.
3. Profile — an agreed rule set for one job — a specification such as A2DP or HFP guaranteeing cross-vendor interoperability.
4. Pairing — the one-time ceremony creating a shared secret — Secure Simple Pairing with ECDH and one of four association models.
5. HCI — the boundary between chip and operating system — host controller interface over UART, USB or SDIO.

31.11 Bluetooth Low Energy

31.11.1 PLAIN - in simple words

1. Bluetooth Low Energy shares the name and the band with classic Bluetooth and almost nothing else. It is a separate protocol.
2. It was created for devices that send tiny amounts of data occasionally and must run for a year on a coin cell: a fitness band, a door sensor, a car key.
3. Instead of building a long-lived connection, a device may simply shout a tiny message every so often. That is **advertising**.
4. If more is needed, a scanner can ask to connect, and then the two exchange named pieces of data organized as **services** and **characteristics**.

31.11.2 PLAIN - a picture in your head

1. Classic Bluetooth is a telephone call: you dial, you stay connected, and you pay for the whole time whether you speak or not.
2. BLE is a postcard on a noticeboard. The device pins a small card up every second, then goes back to sleep.
3. If you want more than the card says, you knock on the door, and the device opens a short conversation from a printed menu of what it can tell you.
4. That menu is the GATT database: sections are services, items are characteristics.

Where this comparison breaks: a noticeboard is passive. A BLE advertisement costs real energy each time, which is why the advertising interval is the main battery knob. Longer intervals mean longer life and slower discovery.

31.11.3 PLAIN - a worked example

1. Take a heart rate monitor chest strap. Its GATT database looks like this.

```
Service: Heart Rate            (UUID 0x180D)
  Characteristic: Heart Rate Measurement (0x2A37)
      properties: NOTIFY
      value: flags byte + rate, e.g. 00 4B  -> 75 bpm
  Characteristic: Body Sensor Location    (0x2A38)
      properties: READ
      value: 01  -> chest

Service: Battery Service       (UUID 0x180F)
  Characteristic: Battery Level           (0x2A19)
      properties: READ, NOTIFY
      value: 5B  -> 91 percent
```

2. The phone does not poll for the heart rate. It subscribes once, and the strap sends a notification each time the value changes.
3. Power arithmetic: a CR2032 coin cell holds about 225 milliamp-hours. Advertising once per second at 10 milliamps for 2 milliseconds averages about 20 microamps, plus 2 microamps sleeping. That is over 1,000 days.

31.11.4 PLAIN - what is really happening inside

1. Here is a BLE device from power-on to data flowing.

```
 1. Power on. The chip configures its radio and loads its
    GATT table from flash into memory.
 2. It picks an advertising interval, say 1000 ms.
 3. Every interval it wakes and transmits the same short
    packet on channels 37, 38 and 39 in turn, then sleeps.
 4. A phone scanning hears one of those three channels.
 5. The phone may send a scan request; the device answers
    with a scan response holding extra bytes, e.g. its name.
 6. The user taps connect. The phone sends a connect request
    carrying an access address, a hop increment and a
    connection interval.
 7. Both sides move onto the 37 data channels, hopping by
    the agreed increment.
 8. At each connection interval the peripheral wakes, listens
    for the central's packet, replies, and sleeps again.
 9. The phone discovers services, then characteristics.
10. The phone writes 0x0001 to the heart rate CCCD.
11. The strap now sends a notification every beat, inside
    the normal connection events. No polling.
```

2. Privacy is a real concern. A device advertising the same address forever is trackable, so BLE supports **resolvable private addresses** that change every fifteen minutes and can be linked back only by a device holding the identity resolving key.
3. Beacons are advertisements with a fixed payload and no connection. Apple's iBeacon, from 2013, and Google's Eddystone, from 2015, are two formats.
4. They enable indoor positioning and also silent presence tracking, which is why both Android and iOS gate scanning behind a location permission.

31.11.5 TECHNICAL - the engineer's version

1. BLE was developed by Nokia as Wibree from 2006, adopted by the Bluetooth SIG and published in the Bluetooth 4.0 core specification in June 2010.
2. Radio: 40 channels of 2 MHz from 2402 to 2480 MHz, with GFSK modulation.
3. Speeds are 1 Mbit/s at LE 1M and 2 Mbit/s at LE 2M, added in 5.0, plus a coded PHY at S=2 or S=8 giving four times the range at one eighth the rate.
4. Stack, bottom to top: link layer, L2CAP, then **ATT** (attribute protocol) and **GATT** (generic attribute profile), with **SMP** (security manager protocol) alongside for pairing.
5. ATT is the wire protocol: a flat table of handles, each with a UUID, permissions and a value, with operations read, write, write without response, notify and indicate.
6. GATT is the convention layered on ATT that groups attributes into services and characteristics. It is a profile, not a separate protocol.
7. Since BLE 4.2, data length extension raises the link layer payload from 27 to 251 bytes, roughly tripling throughput.
8. Tools: `bluetoothctl` and `btmon` on Linux, the nRF Connect application on Android and iOS for exploring a GATT table, and a Nordic nRF52 sniffer with Wireshark for over-the-air capture.

31.11.6 WORDS - remember these

1. Advertising — shouting a small packet periodically — unconnected broadcast on channels 37, 38 and 39.
2. GATT — the small named database on a device — generic attribute profile organizing ATT handles into services and characteristics.
3. Characteristic — one named value you can read or subscribe to — an ATT attribute with a UUID, properties and optional descriptors.
4. Resolvable private address — an address that changes to avoid tracking — a rotating address resolvable only with the identity resolving key.

31.12 Bluetooth audio

31.12.1 PLAIN - in simple words

1. Bluetooth audio was poor for a long time, and the reasons explain most of what people complain about.
2. Every device must support one basic method so anything works with anything. That method, SBC, was designed in 2002 to be cheap, not good.
3. Squeezing and unsqueezing takes time. That delay is **latency**, and it is why lips can be out of step with sound when watching video.
4. Worst of all, when you use the microphone the link switches to a different, much narrower mode and quality collapses.
5. That is why music sounds fine until a call starts, and then everything sounds like a 1990s telephone.

31.12.2 PLAIN - a picture in your head

1. Think of a narrow letterbox. Thin envelopes go through quickly.
2. Now you want a two-way conversation through the same letterbox. Half the space must go to your outgoing envelopes.
3. With half the space gone, the incoming book must be squashed far harder, into something barely readable. That is what happens when the microphone opens.

Where this comparison breaks: a letterbox is only about width. The old profile also switched to a completely different, much older codec, so the damage was larger than a simple halving of bandwidth. LE Audio fixes this by using one modern codec in both directions.

31.12.3 PLAIN - a worked example

Codec	Owner	Bitrate	Notes
SBC	Bluetooth SIG	192-345 kbit/s	Mandatory, baseline
AAC	MPEG	128-256 kbit/s	Apple devices default
aptX	Qualcomm	352 kbit/s	Fixed rate, low latency
aptX HD	Qualcomm	576 kbit/s	24-bit capable
LDAC	Sony	330-990 kbit/s	Highest rate, adaptive
LC3	SIG, 2020	160-345 kbit/s	LE Audio, best per bit

1. Latency figures, approximate and highly device dependent: SBC 150 to 250 ms, AAC on Apple devices 100 to 160 ms.
2. aptX Low Latency is 32 to 40 ms, LC3 over LE Audio is 20 to 30 ms, and a wired 3.5 mm jack is under 1 ms.
3. Humans notice audio and video out of step at roughly 45 milliseconds of audio lag. That is why video players delay the picture when Bluetooth is in use, and why gaming over Bluetooth feels wrong.

31.12.4 PLAIN - what is really happening inside

1. Two different profiles handle audio, and that is the root of the problem.
2. **A2DP**, the advanced audio distribution profile, is one-way, high quality and used for music.
3. **HFP**, the hands-free profile, is two-way and used for calls.
4. When a call starts the headset switches from A2DP to HFP. Classic HFP originally offered only the CVSD codec at 8 kHz sampling, telephone quality, later improved to mSBC at 16 kHz, called wideband speech.
5. **LE Audio**, introduced in Bluetooth 5.2 in 2019 and shipping in products from 2022, replaces both. It uses one codec, **LC3**, developed by Fraunhofer IIS and Ericsson, in both directions over new isochronous channels that guarantee timing.
6. LC3 at 160 kbit/s is judged better than SBC at 345 kbit/s in listening tests run by the Bluetooth SIG, so it gives better sound at half the bitrate, which also means longer battery life.
7. **Auracast** is LE Audio's broadcast mode. One transmitter serves unlimited receivers, so a television, an airport gate or a lecture hall can broadcast audio anyone nearby joins, optionally with a password.
8. The Bluetooth SIG announced the Auracast brand in 2022.

31.12.5 TECHNICAL - the engineer's version

1. HFP uses a SCO or eSCO link, which is synchronous and reserves fixed slots.
2. CVSD is 64 kbit/s at 8 kHz; mSBC is a constrained SBC configuration at 16 kHz over eSCO, defined in HFP 1.6 in 2009.
3. There is genuine expert disagreement about whether LDAC and aptX HD are audibly better than well-implemented AAC. Blind listening tests generally fail to separate them at 256 kbit/s and above.
4. Treat "hi-res Bluetooth" as a marketing claim, and codec efficiency at low bitrate as the engineering fact.
5. LE Audio components: the LC3 codec, isochronous channels (CIS for connected, BIS for broadcast), the Basic Audio Profile, and the Public Broadcast Profile for Auracast.
6. Inspect on Linux with `pactl list sinks` or `bluetoothctl` plus PipeWire or BlueZ logs, which name the negotiated codec. On Android, Developer options shows the Bluetooth audio codec, sample rate and bit depth.

31.12.6 WORDS - remember these

1. HFP — the profile for calls — hands-free profile over a synchronous SCO or eSCO link, historically CVSD then mSBC.

2. LC3 — the modern codec of LE Audio — low complexity communication codec, better quality per bit than SBC.
3. LE Audio — the redesign that fixes call quality — Bluetooth 5.2 isochronous channels with LC3 in both directions.
4. Auracast — broadcasting audio to many listeners — LE Audio broadcast isochronous streams, branded by the SIG in 2022.

31.13 Cellular

31.13.1 PLAIN - in simple words

1. WiFi covers a room. Cellular covers a country with the same amount of spectrum, by reusing it over and over.
2. The trick is the **cell**. Divide the land into patches and give each patch a low-power transmitter that covers only that patch.
3. Two patches far enough apart can use the same frequency without hearing each other. That is **frequency reuse**, and it is the whole idea.
4. As you move you leave one cell and enter another, and the network moves your call across without dropping it. That is **handover**.
5. Unlike WiFi, cellular spectrum is licensed and expensive, so the operator controls exactly who transmits and when. There is no random backoff.

31.13.2 PLAIN - a picture in your head

1. Imagine a country covered in hexagonal tiles, like a bathroom floor.
2. Seven colours are enough that no two touching tiles share a colour. Each colour is a set of frequencies.

Where this comparison breaks: real cells are not hexagons and not equal. Coverage is a messy blob shaped by hills and buildings, cells overlap heavily, and modern networks reuse every frequency in every cell, relying on coding and interference coordination instead of colour separation.

31.13.3 PLAIN - a worked example

Gen	Years	Peak rate	What changed
1G	1979-1991	analogue voice	Cellular concept, AMPS/NMT
2G	1991	9.6 kbit/s	Digital voice, SMS, SIM
2.5G	2000-2003	56-384 kbit/s	Packet data: GPRS, EDGE
3G	2001	384 kbit/s-42 Mbit/s	UMTS, CDMA, mobile web
4G	2009	100 Mbit/s-1 Gbit/s	All-IP, OFDMA, no circuits
5G	2019	20 Gbit/s peak spec	mmWave, slicing, low latency

1. GSM, the dominant 2G system, was agreed by 15 representatives from 13 European countries in Copenhagen in 1987 and launched commercially in Finland in December 1991, with the first call on 1 July 1991.
2. GSM voice used a 13 kbit/s full-rate codec and 9.6 kbit/s circuit-switched data. GPRS services launched in 2000 and EDGE first ran in 2003.
3. 5G frequency ranges: FR1 is everything below 7.125 GHz with channels up to 100 MHz; FR2 is 24.25 to 71 GHz with channels up to 400 MHz.

31.13.4 PLAIN - what is really happening inside

1. The SIM card is a small computer, not a memory card. It runs a program and holds a secret it will never reveal.
2. It holds the ICCID, the card's serial number; the IMSI, which identifies the subscriber; and Ki, a 128-bit secret key.

3. The **IMEI** is different and lives in the phone, not the SIM. It identifies the handset, and is what a stolen-phone blocklist uses.
4. Authentication never sends the key. The network sends a random challenge, the SIM computes a response using Ki, and returns it. The network computes the same response independently, and if they match you are in.
5. In 2G this was one-way: the phone proved itself to the network but not the reverse, which is why fake base stations, IMSI catchers, worked. 3G onwards added mutual authentication.
6. Roaming works because your home network holds Ki and the visited network does not. The visited network asks your home network for challenge and response pairs, and uses them without learning the key.

```
What happens when a phone attaches
1. Power on. Radio scans bands for the strongest cell.
2. It reads broadcast information: operator, cell ID,
   tracking area, and how to send a request.
3. It sends a random-access preamble on a shared channel.
4. The network grants a small uplink allocation.
5. The phone sends an attach request with an identity,
   ideally a temporary one, not the IMSI.
6. Authentication challenge and response, both directions.
7. Ciphering and integrity protection are switched on.
8. The core network sets up a default data bearer.
9. An IP address is assigned, usually behind carrier NAT.
10. Data flows. The phone is paged in its tracking area.
```

7. Now tie this back to the reader's home broadband from Chapter 29. On home broadband your router gets an address and your packets go straight into the ISP's network.
8. On mobile, packets are first carried inside a tunnel from the tower all the way to a gateway deep in the operator's core, and only there do they enter the internet.
9. It is also why the reader's own session showed `github.com` loading instantly on mobile data while timing out on home broadband: the two paths into the internet were entirely different networks with different routing.

31.13.5 TECHNICAL - the engineer's version

1. GSM was specified by ETSI, then everything from 3G onward by 3GPP, which publishes numbered Releases.
2. LTE is Release 8 of 2008; 5G New Radio is Release 15 of 2018, with Release 16 in 2020 adding ultra-reliable low latency and Release 17 in 2022 adding reduced-capability devices.
3. IMSI is 15 digits: a 3-digit mobile country code, a 2 or 3 digit network code, and the subscriber number. India's country codes are 404 and 405. IMEI is 15 digits with a Luhn check digit.
4. Authentication algorithms: 2G used COMP128 variants with A5/1 or the deliberately weakened A5/2 for ciphering, both broken. 3G, 4G and 5G use MILENAGE or TUAK with AES, giving mutual authentication.
5. 5G added SUCI, the subscription concealed identifier, which encrypts the subscriber identity with the operator's public key, closing the IMSI-catcher hole that persisted through 4G.
6. **Network slicing** is a Release 15 feature: several logical networks with separate quality guarantees run over one physical network, identified by an S-NSSAI.
7. Slicing is deployed commercially, but its widely promoted use cases remain partly a marketing claim rather than a mass reality as of 2026.

31.13.6 WORDS - remember these

1. SIM — the small computer holding your identity — a UICC storing IMSI and Ki and computing authentication responses.

2. IMSI — who the subscriber is — 15-digit identity, concealed as SUCI in 5G.
3. IMEI — which handset it is — 15-digit equipment identity with a Luhn check.
4. VoLTE — voice carried as data — SIP over IMS on a dedicated LTE bearer.
5. Network slicing — several virtual networks on one physical one — 3GPP Release 15 S-NSSAI-identified logical networks.

31.14 GPS and positioning

31.14.1 PLAIN - in simple words

1. GPS satellites do not know where you are and never hear from you. They only broadcast. Your receiver does all the work.
2. Each satellite broadcasts the same thing over and over: "I am satellite number seven, here is exactly where I am, and the time is now".
3. Several arrive at slightly different times, because each travelled a different distance. From the delay, the receiver works out each distance.
4. Distance to one satellite puts you somewhere on a sphere around it. Two narrow it to a circle. Three narrow it to two points, one of them absurd, out in space.
5. So three would be enough if your clock were perfect. It is not.
6. Your phone's clock is out by thousandths of a second, and light travels 300 kilometres in a thousandth of a second. That error is enormous.
7. So the receiver treats its own clock error as a fourth unknown and solves for it. Four unknowns need four measurements, so you need four satellites.

31.14.2 PLAIN - a picture in your head

1. Imagine standing in fog with four lighthouses, each sounding a horn at exactly noon.
2. You hear them at slightly different moments, and since sound takes time, the delay tells you the distance to each. If your watch is right, three horns fix your position.
3. If your watch is wrong by a fixed amount, all four distances are wrong by the same amount. That common error is a fingerprint you can solve for.
4. You find the one position and one clock correction that make all four distances consistent. That is trilateration with a clock unknown.

Where this comparison breaks: sound is slow and the satellites move at about 3.9 km per second, so the receiver must also account for satellite motion, the rotation of the Earth during the signal's travel time, and the slowing of radio by the ionosphere.

31.14.3 PLAIN - a worked example

Property	Value
Satellites	24 baseline, 31 usually active
Orbital planes	6, inclined 55 degrees
Altitude	about 20,200 km
Orbital period	11 h 58 min
Civil signal L1	1575.42 MHz
C/A code rate	1.023 Mchip/s

1. Relativity: the satellite clock runs slow by about 7 microseconds a day due to its speed, and fast by about 45 microseconds a day due to weaker gravity. Net, it runs about 38 microseconds per day fast.
2. Uncorrected, that is 38 microseconds times 300 metres per microsecond, about 11 kilometres of drift per day. GPS is engineering that depends on general relativity to work at all.

Start type	What is known	Time to fix
Cold	Nothing	30 s to 15 min
Warm	Almanac, rough position	20 to 45 s
Hot	Recent ephemeris, time	1 to 5 s
Assisted	Data via mobile network	under 2 s

31.14.4 PLAIN - what is really happening inside

1. The GPS signal arrives weaker than the background noise, around -130 dBm against a noise floor near -111 dBm in the same bandwidth. You cannot see it on a spectrum analyser.
2. It is recovered by correlation. Each satellite modulates its signal with a unique 1023-chip code that repeats every millisecond.
3. The receiver generates the same code locally and slides it against the received noise, looking for the offset where the two line up.
4. When they line up, the wanted signal adds coherently over 1023 chips and the noise does not, giving roughly 30 dB of processing gain and lifting the signal above the noise.
5. Once locked, the receiver reads the navigation message at 50 bits per second. A full frame takes 30 seconds; the complete almanac takes 12.5 minutes.
6. That is the real reason a cold start can take 15 minutes. It is waiting to download data at 50 bits per second, not struggling to hear.
7. **Assisted GPS** delivers that same data over the mobile network in milliseconds, plus a rough position and time, so the search space shrinks.
8. A phone rarely uses GPS alone. It fuses satellites, WiFi access point locations from a crowdsourced database, cell tower positions, the accelerometer and gyroscope for dead reckoning, and the barometer for floor level.
9. Indoors the WiFi database usually dominates, which is why a phone knows which shop you are in with no satellite in sight.

31.14.5 TECHNICAL - the engineer's version

1. The first GPS prototype launched in 1978. The full 24-satellite constellation became operational in 1993, and Full Operational Capability was declared in April 1995.
2. Selective Availability, the deliberate degradation of civil accuracy to about 100 metres, was switched off on 1 May 2000, immediately improving civilian accuracy to roughly 10 metres.

System	Country	Satellites	Notes
GLONASS	Russia	24	Operational since 1995
Galileo	EU	24 plus spares	Services from 2016
BeiDou-3	China	30	Global from 2020
NavIC	India	7	Regional, from 2018

3. NavIC, run by ISRO and formerly called IRNSS, covers India and about 1,500 km beyond, uses L5 and S-band, and is mandated for some vehicle tracking and emergency use in India.

31.14.6 WORDS - remember these

1. Trilateration — finding a position from distances — solving for x, y, z from ranges to several known points.
2. Assisted GPS — getting satellite data over the network — A-GPS delivering ephemeris, time and rough position.

3. Processing gain — how a signal below the noise is recovered — about 30 dB from correlating against the 1023-chip C/A code.
4. GNSS — the general name for all such systems — GPS, GLONASS, Galileo, BeiDou and regional NavIC.

31.15 The other short-range radios

31.15.1 PLAIN - in simple words

1. WiFi and Bluetooth are not the only radios in your life. Several others sit in niches where those two are wrong.
2. **NFC** works over four centimetres and can power the other device: a card with no battery works because the reader's field energizes it.
3. **RFID** is the same family, used for shop tags, warehouse pallets and animal microchips.
4. **Zigbee**, **Z-Wave** and **Thread** are for home sensors: tiny messages, years of battery, and a mesh so distant devices relay for each other.
5. **Matter** is not a radio at all. It is a common language spoken over Thread, WiFi and Ethernet so devices from different makers work together.
6. **LoRa** trades speed for distance: a few hundred bits per second, but kilometres of range through buildings.
7. **UWB** measures distance very precisely by timing a very short pulse, to within about ten centimetres.

31.15.2 PLAIN - a picture in your head

1. Think of a toolbox. WiFi is the power drill: fast, hungry, plugged in. Bluetooth is the screwdriver: handy, personal, battery powered.
2. NFC is the key you touch to a lock: no power of its own, works only when almost touching.
3. LoRa is a foghorn: barely any information, but heard for miles. UWB is a tape measure.

Where this comparison breaks: tools do not interfere with each other. Zigbee, Thread, WiFi and Bluetooth all share 2.4 GHz and do interfere, which is why Zigbee channels are chosen to sit in the gaps between WiFi channels 1, 6 and 11, and why a busy WiFi network can make smart bulbs unreliable.

31.15.3 PLAIN - a worked example

Technology	Band	Rate	Range
NFC	13.56 MHz	106-424 kbit/s	under 10 cm
Zigbee	2.4 GHz	250 kbit/s	10-100 m mesh
Z-Wave	865-926 MHz	100 kbit/s	30-100 m mesh
Thread	2.4 GHz	250 kbit/s	10-100 m mesh
LoRa	865-928 MHz	0.3-27 kbit/s	5-15 km
UWB	3.1-10.6 GHz	up to 27 Mbit/s	10-50 m
BLE	2.4 GHz	0.125-2 Mbit/s	10-100 m

1. NFC data rates are 106, 212 or 424 kbit/s under ISO/IEC 18092.

31.15.4 PLAIN - what is really happening inside

1. NFC is not radiation. It is transformer action across a gap.
2. The reader drives a coil at 13.56 MHz, creating a strong alternating magnetic field in the near field, within a few centimetres.
3. A tag's coil sits in that field, and the changing field induces a current that powers the tag's tiny chip.
4. The tag replies not by transmitting but by changing how much energy it absorbs, which the reader detects as a change in its own coil's load. That is **load modulation**.

5. That is why an NFC card works with no battery, and why the range is physically limited: near-field magnetic strength falls off as the cube of distance, not the square.
6. Zigbee and Thread both run on the IEEE 802.15.4 radio, a standard from 2003 defining 250 kbit/s at 2.4 GHz using offset QPSK.
7. Zigbee, from the Zigbee Alliance formed in 2002 with its first specification in 2004, defines its own application layer above that radio.
8. Thread, launched by the Thread Group in July 2014, keeps 802.15.4 but runs IPv6 over it using 6LoWPAN header compression, so every device is a real IP host with no translating gateway.
9. Matter, from the Connectivity Standards Alliance, released version 1.0 on 4 October 2022. It began in December 2019 as Project Connected Home over IP, founded by Amazon, Apple and the Zigbee Alliance.
10. LoRa uses chirp spread spectrum: each symbol is a sweep across the band rather than a fixed tone, and a sweep is easy to detect far below the noise floor.
11. The **spreading factor**, 5 to 12, sets how long each chirp lasts, trading range against data rate.
12. UWB sends pulses under two nanoseconds long, spread over at least 500 MHz. Because the pulse is so short, its arrival time can be measured very precisely, which is what gives centimetre ranging.

31.15.5 TECHNICAL - the engineer's version

1. NFC standards: ISO/IEC 18092 and ECMA-340 define NFCIP-1, ISO/IEC 14443 covers proximity cards, and ISO/IEC 15693 covers vicinity cards at about one metre. The NFC Forum was founded on 18 March 2004 by Nokia, Philips and Sony.
2. RFID spans several bands: 125 to 134 kHz for animal tags, 13.56 MHz for smart cards, and 860 to 960 MHz for retail and logistics, where passive tags read at up to about 10 metres by backscatter.
3. Z-Wave was created by Zensys in Denmark around 1999, acquired by Sigma Designs and then by Silicon Labs in 2018.
4. Its band is region-specific: 908.42 MHz in the United States, 868.42 MHz in Europe, 865.2 MHz in India. Being sub-gigahertz, it avoids 2.4 GHz congestion entirely.
5. UWB was authorized for unlicensed use by the FCC in February 2002 from 3.1 to 10.6 GHz, with a power spectral density limit of -41.3 dBm/MHz.
6. The standards are IEEE 802.15.4a and the newer 802.15.4z, which added secure ranging with scrambled timestamp sequences.
7. UWB ranging uses two-way time of flight, so a clock offset between the two devices cancels. Ten centimetre accuracy is routine.
8. Car makers use UWB for digital keys under the Car Connectivity Consortium Digital Key 3.0 specification, because ranging defeats the relay attacks that break Bluetooth and radio key fobs.
9. LPWAN alternatives to LoRa: NB-IoT and LTE-M are 3GPP standards from Release 13 in 2016, running in licensed cellular spectrum, with NB-IoT using a 180 kHz carrier.
10. Sigfox was a proprietary ultra-narrowband system from a French company founded in 2010; it entered receivership in 2022, which is a useful caution about proprietary networks.

31.15.6 WORDS - remember these

1. NFC — a tap-range link that powers the other device — 13.56 MHz inductive coupling with load modulation, ISO/IEC 18092.
2. RFID — tagging things with a chip and no battery — passive backscatter tags at 125 kHz, 13.56 MHz or 860-960 MHz.
3. Zigbee — a mesh for home sensors with its own application layer — IEEE 802.15.4 radio plus the Zigbee stack, from 2004.
4. Thread — an IPv6 mesh for home sensors — IEEE 802.15.4 plus 6LoWPAN, from the Thread Group in 2014.
5. Matter — one language so smart devices interoperate — a CSA application layer over IP, version 1.0 released October 2022.

6. LoRa — very long range at very low data rates — chirp spread spectrum with spreading factors 5 to 12.
7. UWB — measuring distance to a few centimetres — sub-nanosecond pulses over 500 MHz or more, IEEE 802.15.4z two-way ranging.

31.98 Common wrong ideas

1. Wrong: WiFi and the internet are the same thing. Right: WiFi is only the last few metres of radio. A working WiFi link with a broken uplink gives full signal bars and no internet at all.
2. Wrong: more bars means faster. Right: bars show received signal strength, only one input. A strong signal on a congested channel is slower than a weaker signal on a clear one.
3. Wrong: the advertised router speed is what you get. Right: an "AX5400" figure adds every band and stream together. Real single-client throughput is typically 40 to 60 percent of one band's headline number.
4. Wrong: 5 GHz is always better than 2.4 GHz. Right: 5 GHz carries more data but loses about 6.5 dB more over distance and roughly twice as much through each wall. Through two brick walls, 2.4 GHz usually wins.
5. Wrong: your WiFi password is sent to the router. Right: it never leaves your device. Both sides derive the same key from it, and only random numbers and proofs cross the air.
6. Wrong: Bluetooth and Bluetooth Low Energy are versions of one protocol. Right: they share a brand, a band and a chip, and are otherwise different protocols with different radios, packet formats and stacks.
7. Wrong: Hedy Lamarr invented WiFi. Right: she and George Antheil patented frequency hopping in 1942, an idea Bluetooth uses and WiFi mostly does not.
8. Wrong: GPS satellites track your phone. Right: satellites only broadcast, and a GPS receiver transmits nothing. Anything tracking you does so through the mobile or WiFi network, not through GPS.

31.99 Chapter summary in 20 lines

1. A radio wave is made when alternating current in a conductor sheds a self-sustaining pair of electric and magnetic fields into space.
2. Wavelength is the speed of light divided by frequency: 12.3 cm at 2.4 GHz and 5.8 cm at 5 GHz, which is why antennas are the sizes they are.
3. Radio is light at a frequency you cannot see, and its photons are far too weak to break chemical bonds.
4. Information rides on a wave by changing amplitude, frequency or phase, and modern systems change all of them at once as QAM.
5. More bits per symbol needs a cleaner signal: 1024-QAM needs about 35 dB signal-to-noise ratio where BPSK needs 2 dB.
6. Spread spectrum trades bandwidth for robustness, by hopping in Bluetooth or by spreading with a code in 802.11b and GPS.
7. OFDM sends many slow narrow subcarriers instead of one fast wide one, and its guard interval swallows echoes. It underlies WiFi, LTE and 5G.
8. Signal is measured in dBm against a noise floor near -96 dBm in a 20 MHz channel, and the gap between them decides your data rate.
9. WiFi exists because the FCC opened the ISM bands to unlicensed spread spectrum in 1985, and because ALOHAnet in 1971 had shown how to share a channel by random access.
10. The standards run from 802.11-1997 at 2 Mbit/s to 802.11be in 2024, with Wi-Fi 8 as 802.11bn expected around 2028 aiming at reliability, not speed.
11. CSIRO's US patent 5,487,069 of 1996 covers a real technique WiFi uses and earned about 430 million dollars, but WiFi has no single inventor.
12. On the air, devices listen, wait a random backoff, transmit and wait for an acknowledgement, because a transmitter cannot hear a collision.

13. Overheads mean real throughput is roughly half the advertised rate; aggregation, OFDMA and MU-MIMO exist to claw that back.
14. WEP failed because a 24-bit initialization vector repeats within minutes, WPA2 fixed it with AES-CCMP, KRACK broke the handshake's assumptions in 2017, and WPA3 with SAE removed offline password guessing.
15. In software, firmware, driver, cfg80211 and mac80211, a supplicant and a network manager cooperate, and your password becomes a key locally.
16. Bluetooth began at Ericsson in 1994 with Jaap Haartsen, was published by a five-company SIG in 1998, and hops 1600 times a second across 79 channels.
17. Bluetooth Low Energy is a separate protocol using advertising, connections and a small named database called GATT, designed for coin cells.
18. Cellular reuses the same spectrum in every cell, keeps tight central control, and authenticates with a secret the SIM never reveals.
19. GPS needs four satellites because the receiver's clock is a fourth unknown, and it depends on a relativistic correction of 38 microseconds per day.
20. NFC, RFID, Zigbee, Thread, Matter, LoRa and UWB each exist because WiFi and Bluetooth are the wrong tool for some job.

32

CHAPTER

KB

TLS, HTTPS and the Web Stack

32.0 What this chapter gives you

1. You will be able to write a complete HTTP request by hand, byte by byte, and read a real response header block without guessing at any line.
2. You will be able to name the three separate problems that TLS solves, and say why solving only the first one is close to useless.
3. You will be able to explain symmetric encryption, hashing, message authentication, public keys and key exchange in plain words, and then work a Diffie-Hellman exchange by hand with small numbers.
4. You will be able to walk through both the TLS 1.2 and the TLS 1.3 handshake message by message, and say exactly what TLS 1.3 removed.
5. You will be able to read an X.509 certificate field by field with `openssl`, and say what every field is for.
6. You will be able to explain the chain of trust from leaf to intermediate to root, say where the root store lives on each operating system, and say who decides what goes into it.
7. You will be able to state precisely what a browser checks before it shows the padlock, and, more importantly, the long list of things it does not check.
8. You will be able to explain why accepting a self-signed certificate on the reader's own router at `192.168.0.1` is a reasonable risk, and why accepting the identical warning on a public site destroys everything TLS was for.
9. You will be able to spot a company proxy or an antivirus product opening your encrypted traffic, using one command and one line of its output.
10. You will be able to describe HTTP/1.1, HTTP/2 and HTTP/3 and say what problem each new version was built to fix.

32.1 HTTP first, because TLS wraps it

32.1.1 PLAIN – in simple words

1. HTTP is the language a browser and a web server use to talk.
2. It is text. Real, readable, typed-out text. Not a secret binary format.
3. The conversation has exactly two parts: you ask, the server answers.
4. The thing you send is called a **request**. The thing that comes back is called a **response**.
5. A request says: which verb, which path, which version, then some labelled facts, then maybe some data.
6. A response says: which version, a three-digit number saying how it went, then some labelled facts, then maybe some data.
7. The labelled facts are called **headers**. Each is one line, of the form `Name: value`.
8. The data at the end is called the **body**. A request often has no body. A response usually has one.
9. So to understand HTTPS you must first be able to read HTTP. That is what this section does.

32.1.2 PLAIN - a picture in your head

1. Think of posting a form to a large government office.
2. On the envelope you write the department and the reference number. That is the request line: the verb and the path.
3. Inside, the first page is a cover sheet of small boxes. Your name, your language, the date, whether you want a reply by post. Those are the headers.
4. After the cover sheet come the actual pages you are submitting. That is the body.
5. The office writes back. Their first line is a stamp: APPROVED, REJECTED, MOVED TO ANOTHER OFFICE. That is the status code.
6. Under the stamp is their own cover sheet of boxes, then their pages. Where this comparison breaks: a real office keeps a file on you. HTTP keeps nothing. The server may keep a file, but the protocol itself has no memory at all, and that single fact is the reason cookies had to be invented. Also, a government office handles one form at a time. A modern browser fires off dozens of these exchanges at once over a single connection.

32.1.3 PLAIN - a worked example

1. Here is a real request, typed by hand and sent to `github.com`. The command that sent it is shown further down.
2. Every line ends with two invisible characters, carriage return and line feed, written `\r\n`. The header block ends with one empty line.

```
GET /robots.txt HTTP/1.1\r\n
Host: github.com\r\n
User-Agent: kedbyte/1.0\r\n
Accept: */*\r\n
Connection: close\r\n
\r\n
```

3. Line 1 is the **request line**: verb `GET`, path `/robots.txt`, version `HTTP/1.1`.
4. `Host` says which site you want. One server may host a thousand sites on one address, so this line is compulsory in HTTP/1.1.
5. `User-Agent` says which program is asking. `Accept` says which formats you can read. `Connection: close` asks the server to hang up after replying.
6. The empty line is the end of the headers. Anything after it would be body. A `GET` has no body, so there is nothing after it.
7. Here is the real response that came back, trimmed only where marked.

```
HTTP/1.1 200 OK
date: Thu, 13 Aug 2026 02:28:27 GMT
content-type: text/plain
last-modified: Thu, 13 Aug 2026 02:17:47 GMT
etag: W/"6a7d294b-8e2"
vary: Accept-Encoding, Accept, X-Requested-With
server: github.com
x-frame-options: DENY
strict-transport-security: max-age=31536000;
  includeSubDomains; preload
accept-ranges: bytes
set-cookie: _octo=GH1.1.1717559084.1786588114;
  expires=Fri, 13 Aug 2027 02:28:34 GMT; domain=.github.com;
  path=/; secure; SameSite=Lax
content-length: 2274
x-github-request-id: 4003:3C7715:235C0C:2CA82C:6A7D2BCD
x-github-edge-region: iad
```

```
# If you would like to crawl GitHub contact us via ...
User-agent: bingbot
Disallow: /ekansa/Open-Context-Data
```

8. Line 1 is the **status line**: version `HTTP/1.1`, code `200`, and the human word `OK`.
9. `content-length: 2274` says the body is exactly 2274 bytes long. The reader knows when to stop reading.

32.1.4 PLAIN - what is really happening inside

1. The browser has already opened a TCP connection, which is a two-way stream of bytes to the server. Chapter 30 covers how.
2. HTTP is only an agreement about what to write into that stream.
3. The client writes the request bytes and then waits.
4. The server reads until it sees the blank line, works out what is wanted, and writes the response bytes back.
5. The server must tell the client where the body ends. There are only two ways: give a `Content-Length`, or use `Transfer-Encoding: chunked`, where each piece is preceded by its own size in hexadecimal.
6. Nothing in HTTP is encrypted. Anyone who can read the TCP stream can read every byte shown above, including any password in the body.
7. HTTPS changes exactly one thing: those same bytes are handed to a TLS layer, which encrypts them before they reach TCP, and decrypts them at the far end.
8. The verb, the path, the headers, the cookies and the body are all inside the encryption. The IP address and the port number are not, because the network needs them to deliver anything at all.

32.1.5 TECHNICAL - the engineer's version

1. HTTP/1.0 is RFC 1945, May 1996. HTTP/1.1 was first RFC 2068 in January 1997, then RFC 2616 in June 1999, then split into RFC 7230 to RFC 7235 in June 2014.
2. The current specification is the June 2022 trio: RFC 9110 HTTP Semantics, RFC 9111 HTTP Caching, RFC 9112 HTTP/1.1. Quote these, not RFC 2616.
3. A method is **safe** if it is not meant to change anything on the server. A method is **idempotent** if doing it five times leaves the same state as doing it once.

Method	Safe	Idempotent	Body
GET	yes	yes	no
HEAD	yes	yes	no
OPTIONS	yes	yes	rare
PUT	no	yes	yes
DELETE	no	yes	rare
POST	no	no	yes
PATCH	no	no	yes

4. `GET` fetches. `HEAD` fetches only the headers, so you can check size or freshness without downloading the body.
5. `PUT` replaces a resource at a known path with what you send. `POST` submits data to a handler and the server decides what to create.
6. `PATCH` applies a partial change and is defined in RFC 5789, March 2010. It is not idempotent in general because a patch may be relative.
7. `OPTIONS` asks what is allowed. Its main real use today is the CORS preflight, covered in section 32.11.
8. Status codes come in five families.

Family	Meaning	Example
1xx	Interim, keep waiting	101, 103
2xx	It worked	200, 201, 204
3xx	Go somewhere else	301, 304, 307
4xx	Your request is wrong	400, 403, 404
5xx	Our server is wrong	500, 502, 503

9. The codes actually worth memorizing: 200 OK, 201 Created, 204 No Content, 206 Partial Content, 301 Moved Permanently, 302 Found, 304 Not Modified, 307 Temporary Redirect, 308 Permanent Redirect, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 405 Method Not Allowed, 409 Conflict, 429 Too Many Requests, 500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable, 504 Gateway Timeout.
10. 401 means "you have not proved who you are". 403 means "I know who you are and you still may not". People mix these up constantly.
11. Request headers that matter: `Host`, `User-Agent`, `Accept`, `Accept-Encoding`, `Authorization`, `Cookie`, `Referer` (misspelled in the original 1996 specification and never fixed), `If-None-Match`, `Range`, `Origin`, `Content-Type`, `Content-Length`.
12. This is the exact command used to produce the real output above. It opens a TLS connection and then types HTTP into it by hand.

```
printf 'GET /robots.txt HTTP/1.1\r\nHost: github.com\r\n\
Connection: close\r\n\r\n' \
| openssl s_client -quiet -connect github.com:443 \
    -servername github.com
```

32.1.6 WORDS - remember these

1. HTTP — the ask-and-answer language of the web — a stateless application protocol defined by RFC 9110 to RFC 9112, 2022.
2. Request line — the first line you send — method, request-target and HTTP-version separated by single spaces.
3. Header — one labelled fact per line — a case-insensitive field name, a colon, optional whitespace, and a field value.
4. Body — the actual data — the payload delimited by `Content-Length` or by chunked transfer encoding.
5. Safe method — does not change anything — a method with no intended state change on the origin server, so it may be cached or prefetched.

32.2 The three problems TLS solves

32.2.1 PLAIN - in simple words

1. TLS stands for Transport Layer Security. It is the lock on HTTPS.
2. People think it solves one problem, secrecy. It actually solves three, and the third is the one that matters most.
3. **Confidentiality**: nobody in the middle can read what you send.
4. **Integrity**: nobody in the middle can change what you send without being caught.
5. **Authenticity**: you are really talking to the machine that owns the name you typed, and not to somebody pretending.
6. Confidentiality without authenticity is nearly worthless.
7. Here is why, in one line: if an attacker sits in the middle and you set up a perfectly encrypted tunnel to the attacker, your secrets are encrypted all the way into the attacker's hands.
8. So the first job of TLS is not encryption. It is proving who the other end is. Encryption only becomes useful after that proof.

32.2.2 PLAIN – a picture in your head

1. Imagine sending a valuable parcel to a friend in another city.
2. Confidentiality is the sealed opaque box. Nobody along the way can see what is inside.
3. Integrity is the tamper-evident seal. If somebody opens the box and swaps the contents, the broken seal shows it.
4. Authenticity is checking the identity of the person who signs for it at the other end.
5. Now picture a courier who quietly redirects your parcel to his own flat, signs for it himself, opens it, and repacks it in an identical box with a fresh seal before sending it on.
6. The box was sealed. The seal was intact on arrival. And you were robbed anyway, because you never checked who signed for it.
7. That is exactly a man-in-the-middle attack, and it is why authenticity comes first.

Where this comparison breaks: a courier must physically hold your parcel. On a network an attacker only needs to sit on any link along a path that may be thousands of kilometres long. The reader's own traceroute crossed at least three organizations before Pune. Any one of them is a place a box could be opened.

32.2.3 PLAIN – a worked example

1. Take three concrete attacks, one per property.
2. **Attack on confidentiality: passive sniffing.** The reader is on public cafe wireless. An attacker runs a laptop in monitor mode and records every frame in the air. Over plain HTTP the attacker reads the session cookie and logs in as the reader.
3. What defeats it: encryption. The recorded bytes are meaningless without the key.
4. **Attack on integrity: content injection.** An ISP or a hotel network rewrites a plain HTTP page as it passes, adding an advertising script, or swapping a download link for a modified installer. This has been done commercially and by governments.
5. What defeats it: a message authentication tag on every record, so any changed byte makes the receiver throw the whole record away.
6. **Attack on authenticity: impersonation.** An attacker answers the DNS query for `github.com` with his own address, or announces a false route, and your browser connects to his server. He offers you a perfectly good TLS connection to himself.
7. Now combine them. Encryption alone stops attack 2 from being read but not attack 6 from happening. Authentication alone stops attack 6 but leaves attack 2 wide open. You need all three, and TLS gives all three in one protocol.

32.2.4 PLAIN – what is really happening inside

1. TLS sits between the application and TCP. HTTP hands it bytes; it hands TCP encrypted records; TCP hands the network packets.
2. During the handshake the two sides agree on which algorithms to use, prove identity, and jointly create secret keys.
3. Identity is proved by a certificate plus a signature. The server sends a certificate that says "this public key belongs to github.com", and then signs a value derived from the live conversation with the matching private key.
4. That second step is what stops replay. Anybody can copy a certificate. Only the true owner can sign fresh data with the private key inside it.
5. Once identity is settled, the two sides derive symmetric keys and switch to fast bulk encryption for everything after.

32.2.5 TECHNICAL – the engineer's version

1. TLS provides confidentiality and integrity through an AEAD cipher, which means Authenticated Encryption with Associated Data, and authenticity through X.509 certificate-based server authentication.

2. In TLS 1.3, defined in RFC 8446 of August 2018, all cipher suites are AEAD. Non-authenticated modes were removed entirely.
3. Server authentication is mandatory in practice on the web. Client authentication with a client certificate is optional and rare outside enterprises and machine-to-machine APIs.
4. Anonymous key exchange suites exist in the specifications and are disabled everywhere. They give encryption with no authentication, which is the useless case described above.
5. A property TLS does not provide: hiding who you are talking to. The destination IP address is in the clear, and until Encrypted Client Hello the server name was in the clear too.
6. Encrypted Client Hello was published as RFC 9849 in March 2026. It encrypts the server name and the protocol list in the first message. Deployment is uneven; treat it as new rather than universal.

32.2.6 WORDS - remember these

1. Confidentiality — nobody can read it — an attacker with the full ciphertext learns nothing about the plaintext beyond its length.
2. Integrity — nobody can change it undetected — any modification is rejected by an authentication tag check.
3. Authenticity — you are talking to the right machine — the peer proved possession of the private key matching a certified public key.
4. Man in the middle — someone sitting between you and the server — an active on-path attacker who terminates and re-originates the connection.

32.3 The crypto building blocks, taught gently

32.3.1 PLAIN - in simple words

1. TLS is built from five simple ideas. None of them is hard on its own.
2. **Symmetric encryption**: one shared secret scrambles and unscrambles. Fast. Both sides need the same key.
3. **Hashing**: turn any amount of data into a short fixed fingerprint. Easy forwards, impossible backwards.
4. **Message authentication code**: a fingerprint that only somebody with the secret can produce, so it proves the data was not changed by an outsider.
5. **Asymmetric encryption**: two matching keys. One is public, one is private. What one does, only the other can undo.
6. **Key exchange**: a way for two strangers to end up holding the same secret number, even though everything they said was heard by everyone.
7. TLS uses asymmetric maths only at the start, to prove identity and agree a key, because it is slow. Then it uses symmetric encryption for all the real traffic, because it is fast.

32.3.2 PLAIN - a picture in your head

1. For Diffie-Hellman key exchange, use paint.
2. You and a stranger stand at opposite ends of a crowded room. Everyone can see what you carry across.
3. First you both agree, out loud, on a common colour. Say yellow. Everybody hears this. That is fine.
4. You privately pick a secret colour, red. You keep the tin hidden. The stranger privately picks blue and hides it.
5. You mix your red into the yellow and get orange. He mixes his blue into the yellow and gets green.
6. You swap. You hand him orange in public. He hands you green in public. The crowd sees orange and green.
7. Now you add your hidden red to the green you received. He adds his hidden blue to the orange he received.
8. You both hold the same colour: yellow plus red plus blue. The crowd, who only saw yellow, orange and green, cannot make it.

9. The trick is that mixing paint is easy and unmixing it is very hard.

Where this comparison breaks: unmixing paint is only messy, not impossible. The real system uses modular exponentiation, and reversing it means solving the discrete logarithm problem, which nobody can do quickly. Also, the shared result is a number, not a key; it is passed through a key derivation function first. And on its own it tells you nothing about who the stranger is, which is why certificates must be bolted on.

32.3.3 PLAIN – a worked example

1. Let us do the real arithmetic with numbers small enough to check by hand.
2. Public, agreed out loud: a prime `p = 23` and a base `g = 5`. Everyone knows both.
3. Alice secretly picks `a = 6`. Bob secretly picks `b = 15`.
4. Alice computes `A = g^a mod p`, that is `5^6 mod 23`. Now `5^6 = 15625`, and `15625 = 679 * 23 + 8`, so `A = 8`.
5. Bob computes `B = g^b mod p = 5^15 mod 23 = 19`.
6. They send 8 and 19 across the room. An eavesdropper now knows `p = 23`, `g = 5`, `A = 8`, `B = 19`.
7. Alice computes `B^a mod p = 19^6 mod 23 = 2`.
8. Bob computes `A^b mod p = 8^15 mod 23 = 2`.
9. Both hold 2, and it was never sent. This is real output from `python3`:

```
p = 23  g = 5
Alice secret a = 6  -> A = g^a mod p = 8
Bob   secret b = 15 -> B = g^b mod p = 19
Alice computes B^a mod p = 2
Bob   computes A^b mod p = 2
```

10. Why it works: `B^a = (g^b)^a = g^(ba)` and `A^b = (g^a)^b = g^(ab)`, and `ab` equals `ba`, so both are `g^(ab) mod p`.
11. Why the eavesdropper is stuck: to get `a` from `A = 8` he must find which power of 5 gives 8 modulo 23. With `p = 23` he can just try all of them. Here are the powers of 5 modulo 23, in order:

```
5, 2, 10, 4, 20, 8, 17, 16, 11, 9, 22, 18, 21, 13, 19, 3,
15, 6, 7, 12, 14, 1
```

12. The sixth entry is 8, so `a = 6`. Trivial at this size.
13. In real use `p` is 2048 or 3072 bits, so that list has more entries than there are atoms in the observable universe, and there is no known shortcut.
14. Now hashing, with real `openssl` output. Change one letter and the whole fingerprint changes:

```
$ echo -n "hello" | openssl dgst -sha256
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362
938b9824

$ echo -n "hellp" | openssl dgst -sha256
fdd7585e08c4e2afd71dcabdb4636c89d557a3f42db9e2040c8bbd17
08aa4ce7
```

15. Each output is 64 hexadecimal characters, which is 256 bits, which is why it is called SHA-256. Both were wrapped over two lines to fit this page.

32.3.4 PLAIN – what is really happening inside

1. **Symmetric encryption.** AES takes a key of 128 or 256 bits and a block of 16 bytes, and shuffles the block using the key through 10 to 14 rounds of substitution and mixing. The same key run in reverse undoes it.
2. Modern TLS uses AES-GCM, which encrypts and also produces a 16-byte authentication tag over the ciphertext and the record header. One primitive, both properties.

3. Here is a real AES-256-GCM run. Note the tag, and note what happens when one single bit of the ciphertext is flipped:

```
plaintext  = meet me at six   (14 bytes)
ciphertext = aaacf7bb4bf766de898f5ea22006
tag        = 1cee6a97e098c053433578dfe18e3d22
flip one bit -> decrypt raises InvalidTag
honest decrypt -> meet me at six
```

4. **Hashing.** SHA-256 chews the input in 64-byte blocks, keeping eight 32-bit working values that get stirred by each block. The final eight values, joined, are the digest. There is no key and no way back.
5. The property we rely on is **collision resistance**: nobody can find two different inputs with the same digest. When that broke for MD5 in 2004 and for SHA-1 in 2017, both had to be removed from certificates.
6. **Asymmetric.** RSA works with a public pair `(n, e)` and a private exponent `d`. Encrypting is `c = m^e mod n`. Decrypting is `m = c^d mod n`. Signing is the same operation with the private key, and verifying uses the public key.
7. **Signatures in practice.** You never sign the message. You hash it, then sign the hash. That is why every certificate says something like `sha256WithRSAEncryption`: hash with SHA-256, sign with RSA.
8. Here is real signing and verifying, with a genuine 2048-bit RSA key made for this chapter:

```
$ openssl dgst -sha256 -sign ca.key -out msg.sig msg.txt
$ openssl dgst -sha256 -verify ca.pub -signature msg.sig \
    msg.txt
Verified OK

$ openssl dgst -sha256 -verify ca.pub -signature msg.sig \
    bad.txt
rsa routines:ossl_rsa_verify:bad signature
Verification failure
```

32.3.5 TECHNICAL - the engineer's version

1. AES is FIPS 197, published November 2001, from the Rijndael cipher by Joan Daemen and Vincent Rijmen. Block size 128 bits; key sizes 128, 192 and 256 bits; 10, 12 and 14 rounds.
2. HMAC is RFC 2104, February 1997. It is defined as `H((K xor opad) || H((K xor ipad) || M))`, which nests two hashes so that length-extension tricks do not work.
3. RSA was described by Ron Rivest, Adi Shamir and Leonard Adleman in 1977 and published in February 1978. Diffie-Hellman came first, in the November 1976 paper "New Directions in Cryptography" by Whitfield Diffie and Martin Hellman.
4. Elliptic curve cryptography was proposed independently by Neal Koblitz and Victor Miller in 1985. Curve25519 was published by Daniel J. Bernstein in 2006 and is specified for use in TLS by RFC 7748, January 2016.
5. Elliptic curves replaced RSA for new deployments because they give the same security with far smaller keys and far cheaper operations. Here are real key sizes measured on this machine:

Key	Private PEM	Public DER
RSA 2048	1708 bytes	294 bytes
RSA 3072	2488 bytes	422 bytes
P-256	227 bytes	91 bytes
P-384	288 bytes	120 bytes

6. **Forward secrecy** means that recording today's traffic and stealing the server's private key tomorrow does not let the attacker decrypt what was recorded.

7. It works because the actual session key comes from an **ephemeral** Diffie-Hellman exchange, with a key pair generated for that one connection and then thrown away. The long-term private key only signs; it never encrypts the session key.
8. Old RSA key transport, where the client encrypted the premaster secret to the server's RSA public key, had no forward secrecy. That is precisely why TLS 1.3 removed it. Every TLS 1.3 handshake is forward secret.
9. You can see the ephemeral key in real output. This line from a live handshake to `github.com` shows an ephemeral X25519 key was used:

```
Server Temp Key: X25519, 253 bits
New, TLSv1.2, Cipher is ECDHE-RSA-AES256-GCM-SHA384
```

10. The `E` at the end of `ECDHE` is the important letter. It stands for ephemeral. `ECDH` without the `E` is static and not forward secret.
11. Chapter 44 goes much further into cryptography, including post-quantum key exchange, which is now being deployed in TLS to protect against future quantum computers recording traffic today.

32.3.6 WORDS - remember these

1. Symmetric encryption — one shared key both ways — a block or stream cipher such as AES-256-GCM, keyed identically at both ends.
2. Hash — a short fingerprint of any data — a one-way function such as SHA-256 producing a fixed 256-bit digest.
3. Public key — the half you publish — the key used to verify signatures and, in RSA, to encrypt to the holder.
4. Private key — the half you never share — the key used to sign and to decrypt, and the only thing that proves identity.
5. Diffie-Hellman — how strangers agree a secret in public — a key agreement using modular exponentiation or elliptic curve point multiplication.
6. Ephemeral — thrown away after one use — a per-connection key pair, giving forward secrecy, marked by the `E` in `ECDHE`.

32.4 The TLS handshake, in full

32.4.1 PLAIN - in simple words

1. Before any web page byte moves, the two machines have a short conversation to set up the lock. That conversation is the **handshake**.
2. It does four jobs: agree the version, agree the algorithms, prove the server's identity, and create the shared keys.
3. In the old design this took two full round trips before you could send your first HTTP request.
4. A round trip is one message out and one message back. Over a long path that costs real time, and the reader's own path went from India to Microsoft edge sites in Delhi, Mumbai and Pune.
5. The current design, TLS 1.3, does the same four jobs in one round trip.
6. It manages this by guessing. The client sends its key-exchange material in the very first message, assuming the server will accept it. Almost always the guess is right.
7. There is also a mode with zero round trips, for repeat visits. It is faster and it has a real weakness, described later in this section.

32.4.2 PLAIN - a picture in your head

1. Think of two people meeting to do business, who have never met before.
2. TLS 1.2 is the polite, slow version. "Hello, I speak English, French and Hindi." "Hello, let us use English." "Here is my identity card." "Here is a number I have signed to prove the card is mine." "Now here is my half of the secret." "Here is my half." "Agreed, switching to code." "Agreed."

3. Two complete there-and-back exchanges before a single word of business.
4. TLS 1.3 is the version where you have done this a hundred times before and you know how it will go. "Hello, I speak English, and here is my half of the secret already, assuming you agree." "Agreed, here is mine, here is my card, here is my signature, and I am already speaking in code."
5. One there-and-back. The business starts on your second message. Where this comparison breaks: people can recognize a face. Machines cannot, so the identity card must be checked mathematically every single time. And the "half of the secret" is not half of anything; it is a full public value that is useless without the matching private one.

32.4.3 PLAIN - a worked example

1. Here is a real TLS 1.2 handshake, captured live against `github.com`. The >>> lines were sent by us, the <<< lines came back.

```
>>> TLS 1.2, Handshake, ClientHello
<<< TLS 1.2, Handshake, ServerHello
<<< TLS 1.2, Handshake, Certificate
<<< TLS 1.2, Handshake, ServerKeyExchange
<<< TLS 1.2, Handshake, ServerHelloDone
>>> TLS 1.2, Handshake, ClientKeyExchange
>>> TLS 1.2, ChangeCipherSpec
>>> TLS 1.2, Handshake, Finished
<<< TLS 1.2, Handshake, NewSessionTicket
<<< TLS 1.2, ChangeCipherSpec
<<< TLS 1.2, Handshake, Finished
```

2. Count the direction changes: out, in, out, in. That is two round trips before application data.
3. Now the same server, forced to TLS 1.3:

```
>>> TLS 1.3, Handshake, ClientHello
<<< TLS 1.3, Handshake, ServerHello
<<< TLS 1.3, Handshake, EncryptedExtensions
<<< TLS 1.3, Handshake, Certificate
<<< TLS 1.3, Handshake, CertificateVerify
<<< TLS 1.3, Handshake, Finished
>>> TLS 1.3, ChangeCipherSpec
>>> TLS 1.3, Handshake, Finished
```

4. Out, in, out. One round trip, and the client can attach its HTTP request to that last outbound flight.
5. Notice what disappeared: `ServerKeyExchange`, `ServerHelloDone` and `ClientKeyExchange` are gone. Their work moved into the two hello messages.
6. Notice what appeared: `EncryptedExtensions` and `CertificateVerify`. And notice that from `EncryptedExtensions` onwards, everything is already encrypted, including the server's certificate.
7. The single `ChangeCipherSpec` still there in TLS 1.3 is a fake. It does nothing. It is sent only so that old network middleboxes, which expected to see one, do not break the connection.

32.4.4 PLAIN - what is really happening inside

1. In `ClientHello` the client sends: the highest version it supports, 32 random bytes, a session identifier, a list of cipher suites it will accept, and a list of extensions.
2. Two extensions matter enormously. `server_name` carries the hostname, so one address can serve many sites. `supported_versions` is how TLS 1.3 actually announces itself, because the old version field had to be frozen at 1.2 for compatibility.
3. In TLS 1.3 the client also sends `key_share`: an actual ephemeral public key, usually X25519, guessed in advance.
4. In `ServerHello` the server picks one cipher suite, sends its own 32 random bytes, and in TLS 1.3 sends its own `key_share`.

5. At that instant both sides can compute the shared secret, so everything after the `ServerHello` can be encrypted. That is the core trick of TLS 1.3.
6. `Certificate` carries the server's certificate and the intermediates needed to build a chain. In TLS 1.3 it is encrypted, so a passive observer no longer learns which certificate was served.
7. `CertificateVerify` is a signature, made with the certificate's private key, over a hash of the entire handshake so far. This is the step that proves the server actually holds the private key and is not just replaying a copied certificate.
8. `Finished` is a MAC over the whole transcript, sent by each side. If any earlier message was altered, the two transcripts differ and the check fails. This is what protects the cipher negotiation from downgrade.
9. In TLS 1.2 the equivalent of step 7 was `ServerKeyExchange`, which carried the server's ephemeral DH parameters plus a signature over them. The client then sent `ClientKeyExchange` with its own half.

32.4.5 TECHNICAL - the engineer's version

1. The version history, with real dates:

Version	Published	Reference
SSL 2.0	Feb 1995	Netscape, no RFC
SSL 3.0	1996	RFC 6101 (2011)
TLS 1.0	Jan 1999	RFC 2246
TLS 1.1	Apr 2006	RFC 4346
TLS 1.2	Aug 2008	RFC 5246
TLS 1.3	Aug 2018	RFC 8446

2. SSL 1.0 was never released publicly; it had known flaws. SSL 2.0 shipped in Netscape Navigator 1.1 in 1995. SSL 3.0 was a full redesign by Paul Kocher with Phil Karlton and Alan Freier at Netscape. Taher Elgamal, then Netscape's chief scientist, is widely credited as a driving force behind SSL.
3. TLS 1.3 removed, deliberately: RSA key transport, static Diffie-Hellman, custom DH groups, renegotiation, compression, CBC mode ciphers, all non-AEAD ciphers, RC4, DSA, MD5 and SHA-1 signatures in the handshake, and the change cipher spec protocol as a real message.
4. TLS 1.3 defines only five cipher suites, and OpenSSL 3.0.13 on this machine offers three of them:

```
TLS_AES_256_GCM_SHA384        Enc=AESGCM(256)  Mac=AEAD
TLS_CHACHA20_POLY1305_SHA256  Enc=CHACHA20     Mac=AEAD
TLS_AES_128_GCM_SHA256        Enc=AESGCM(128)  Mac=AEAD
```

5. Reading a TLS 1.2 cipher suite string. Take `ECDHE-RSA-AES256-GCM-SHA384`, which is what `github.com` actually negotiated when forced to TLS 1.2 in the test above.
6. `ECDHE` is the key exchange: elliptic curve Diffie-Hellman, ephemeral. `RSA` is the authentication: the certificate holds an RSA key used for the signature. `AES256` is the bulk cipher and its key size. `GCM` is the mode, giving AEAD. `SHA384` is the hash used by the key derivation function and, in older suites, by the MAC.
7. TLS 1.3 suite names dropped the key exchange and authentication parts entirely, because those are now negotiated by separate extensions. That is why `TLS_AES_128_GCM_SHA256` looks so short.
8. Here are the two handshakes side by side.

```
TLS 1.2 - two round trips before data

Client                              Server
  |-- ClientHello ------------------->|
  |<------------------ ServerHello --|
  |<------------------ Certificate --|
  |<-------------- ServerKeyExchange -|
```

```
   |<---------------- ServerHelloDone -|
   |-- ClientKeyExchange ------------->|
   |-- ChangeCipherSpec -------------->|
   |-- Finished ---------------------->|
   |<---------------- ChangeCipherSpec-|
   |<------------------------ Finished-|
   |-- HTTP GET ---------------------->|

TLS 1.3 - one round trip before data

Client                              Server
   |-- ClientHello + key_share ------->|
   |<-------- ServerHello + key_share -|
   |<------------ EncryptedExtensions -|   encrypted
   |<-------------------- Certificate -|   encrypted
   |<-------------- CertificateVerify -|   encrypted
   |<---------------------- Finished --|   encrypted
   |-- Finished + HTTP GET ----------->|   encrypted
```

9. **0-RTT**, also called early data. On a repeat visit the client has a pre-shared key from a `NewSessionTicket` sent during the previous connection. It can encrypt application data with a key derived from that ticket and put it in the very first flight, alongside the `ClientHello`.
10. The caveat, stated plainly in RFC 8446 section 8 and appendix E.5: 0-RTT data has **no replay protection**. An attacker who records the early-data flight can send it again later, and the server may accept it a second time.
11. Also, 0-RTT data is not forward secret, because it is protected by keys derived from the earlier ticket rather than by a fresh exchange.
12. The practical rule: only allow 0-RTT for safe, idempotent requests. Never for a `POST` that moves money. Cloudflare and other providers restrict early data to `GET` requests without query parameters by default.

32.4.6 WORDS - remember these

1. Handshake — the setup conversation — the sub-protocol that negotiates parameters, authenticates the peer and establishes keys.
2. ClientHello — the opening message — carries versions, random bytes, cipher suites, and extensions including `server_name` and `key_share`.
3. Cipher suite — the chosen set of algorithms — key exchange, authentication, bulk cipher and hash, named as one string.
4. key_share — the client's guess at the key exchange — an ephemeral public key sent in the first message so TLS 1.3 needs only one round trip.
5. CertificateVerify — proof the server owns the key — a signature over the handshake transcript using the certificate's private key.
6. Finished — the tamper check on the whole conversation — a MAC over the full transcript that defeats downgrade attacks.

32.5 Certificates: what is actually in one

32.5.1 PLAIN - in simple words

1. A certificate is a small signed document that says: "this public key belongs to this name".
2. It is not secret. It is sent to everyone who connects. There is nothing private inside it.
3. It contains a name, a public key, two dates, the name of whoever vouched for it, and that voucher's signature.
4. The signature is the whole point. Without it the document is just a claim anybody could type.

5. Your browser does not trust the certificate because of what it says. It trusts it because of who signed it, and because it can check that signature mathematically.
6. The format is called X.509. It is old, it is complicated, and every browser and server on earth uses it.

32.5.2 PLAIN - a picture in your head

1. Think of a passport.
2. The photograph is the public key: the part that identifies you and can be shown to anyone.
3. Your face is the private key: the thing that must match the photograph, that you carry with you, and that cannot be copied out of the book.
4. The name page is the subject: who this passport is for.
5. The issuing authority printed on the cover is the issuer: who vouched.
6. The expiry date is the validity period.
7. The holographic security printing is the signature: hard to forge, easy to check, and it covers the whole book so nothing can be swapped. Where this comparison breaks: a passport is checked by a human who can use judgement. A certificate is checked by software that follows fixed rules and has none. Also, a passport proves who you are. A certificate only proves you control a name. It says nothing about whether you are honest, and section 32.7 makes a great deal of that difference.

32.5.3 PLAIN - a worked example

1. Here is a real certificate, created with `openssl` for this chapter. It is for a home router, and it is used again in section 32.8.
2. It was made with `openssl req -x509` for the authority, then `openssl req` for the request, then `openssl x509 -req -CA ca.crt` to sign. An extensions file supplied the modern half of the certificate:

```
subjectAltName=DNS:router.home,IP:192.168.0.1
basicConstraints=CA:FALSE
keyUsage=digitalSignature,keyEncipherment
extendedKeyUsage=serverAuth
```

3. Now read it back. This is genuine `openssl x509 -text` output, with the long key and signature byte columns removed to fit the page:

```
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: 6EBE14595FF0AED0A54EF7E6925D84514B5421DC
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: C = IN, O = KedByte Home CA,
            CN = KedByte Home Root CA
    Validity
      Not Before: Aug 13 02:27:52 2026 GMT
      Not After : Sep 14 02:27:52 2027 GMT
    Subject: CN = router.home
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
        Public-Key: (2048 bit)
        Exponent: 65537 (0x10001)
    X509v3 extensions:
      X509v3 Subject Alternative Name:
        DNS:router.home, IP Address:192.168.0.1
      X509v3 Basic Constraints:
        CA:FALSE
      X509v3 Key Usage:
        Digital Signature, Key Encipherment
      X509v3 Extended Key Usage:
        TLS Web Server Authentication
```

```
      X509v3 Subject Key Identifier:
        A4:68:C7:64:BF:C2:49:91:9E:73:CE:44:3C:86:19:70:
        52:25:4B:3B
  Signature Algorithm: sha256WithRSAEncryption
```

4. Every one of those lines matters. Read them in order and the whole design becomes obvious.
5. Now the verification, which is the part people never actually try:

```
$ openssl verify router.crt
CN = router.home
error 20 at 0 depth lookup: unable to get local issuer
  certificate
error router.crt: verification failed

$ openssl verify -CAfile ca.crt router.crt
router.crt: OK
```

6. The same certificate failed and then passed. Nothing about the certificate changed. Only the set of trusted issuers changed. That single fact is the entire subject of section 32.6.

32.5.4 PLAIN - what is really happening inside

1. A certificate is a structure written in a binary encoding called DER, then often wrapped in base 64 text between `BEGIN CERTIFICATE` and `END CERTIFICATE` lines, which is called PEM format.
2. The structure has three top-level parts: the body, the algorithm identifier, and the signature bits.
3. Signing means: take the exact bytes of the body, hash them with SHA-256, and then apply the issuer's private key operation to that hash.
4. Verifying means: take the same body bytes, hash them the same way, and use the issuer's public key to check that the signature corresponds to that hash.
5. Because the signature covers the body bytes exactly, changing any single character anywhere in the certificate breaks it. You cannot edit the expiry date. You cannot add a hostname.
6. The **subject** field is the old way of naming the site, using a Common Name. It is dead. Since RFC 9525 of November 2023, the Common Name must not be used for identity at all.
7. The living way is the **Subject Alternative Name** extension, usually shortened to SAN. It is a list, so one certificate can cover `example.com`, `www.example.com` and `*.api.example.com` at once.
8. `basicConstraints: CA:FALSE` says this certificate may not sign other certificates. This is what stops a normal website certificate from being used to mint fake ones. It is checked, and it is critical.

32.5.5 TECHNICAL - the engineer's version

1. X.509 began in 1988 as part of the CCITT X.500 directory series. The profile used on the internet is RFC 5280, May 2008, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile".
2. Field by field, the body of a certificate contains: version, serial number, signature algorithm, issuer distinguished name, validity `notBefore` and `notAfter`, subject distinguished name, subject public key info, and the extensions.
3. Extensions that matter in practice: `subjectAltName`, `basicConstraints`, `keyUsage`, `extendedKeyUsage`, `authorityKeyIdentifier`, `subjectKeyIdentifier`, `crlDistributionPoints`, `authorityInfoAccess` which carries the OCSP responder and issuer URLs, `certificatePolicies`, and the signed certificate timestamp list used by Certificate Transparency.
4. Maximum lifetime is set by the CA/Browser Forum Baseline Requirements, not by any RFC. It was 398 days for certificates issued on or after 1 September 2020.
5. Ballot SC-081v3, passed in April 2025, sets a reducing schedule:

From	Max lifetime
until 14 Mar 2026	398 days
15 Mar 2026	200 days
15 Mar 2027	100 days
15 Mar 2029	47 days

6. Domain validation reuse periods shrink on the same schedule, down to 10 days from March 2029. The direction of travel is short lifetimes and full automation.
7. Here is a real, unmodified root certificate from this machine's trust store, which you can check yourself on any Linux box:

```
$ openssl x509 -in /etc/ssl/certs/ISRG_Root_X1.pem \
    -noout -subject -issuer -dates
subject=C = US, O = Internet Security Research Group,
        CN = ISRG Root X1
issuer=C = US, O = Internet Security Research Group,
        CN = ISRG Root X1
notBefore=Jun  4 11:04:38 2015 GMT
notAfter=Jun  4 11:04:38 2035 GMT
```

8. Subject equals issuer. That is the definition of a self-signed certificate, and every root in every trust store on earth is one.
9. The honest version: the sandbox used to write this chapter sends all outbound traffic through a company proxy that re-issues certificates. So the live chain captured here shows that proxy as the issuer, not a public authority. Section 32.9 uses that as the worked example, because it is a perfect real specimen of interception rather than a flaw in the method.

32.5.6 WORDS - remember these

1. Certificate — a signed statement binding a name to a key — an X.509 version 3 structure profiled by RFC 5280.
2. Subject — who the certificate is about — a distinguished name, now effectively decorative for TLS identity.
3. Issuer — who signed it — the distinguished name of the certificate authority whose private key produced the signature.
4. SAN — the real list of names this certificate covers — the Subject Alternative Name extension, the only source of identity since RFC 9525.
5. Validity — the two dates it works between — `notBefore` and `notAfter`, compared against the verifier's own clock.
6. basicConstraints — whether this may sign other certificates — a critical extension carrying `CA:TRUE` or `CA:FALSE` and an optional path length.

32.6 The chain of trust

32.6.1 PLAIN - in simple words

1. Your browser does not know `github.com`. It has never heard of it.
2. What your browser does know is a list of about 150 organizations it was told to trust, which came with your operating system or your browser.
3. Those are the **root** certificate authorities.
4. A root does not sign website certificates directly. It signs a small number of **intermediate** certificates.
5. Intermediates sign the actual website certificates, called **leaf** certificates.
6. So the shape is: root signs intermediate, intermediate signs leaf, leaf belongs to the website.

7. When you connect, the server sends you the leaf and usually the intermediate. Your browser already has the root.
8. The browser then checks each signature in turn until it reaches something in its own list. If it gets there, the chain is trusted. If it does not, you get a warning.

32.6.2 PLAIN – a picture in your head

1. Think of a university degree certificate.
2. You have never met the person holding it. You cannot verify their study.
3. But the degree is signed by a department head. The department head was appointed by the university. The university is recognized by a national body you already accept.
4. You do not check the student. You check the chain of signatures upward until you reach a body you already decided to accept, and then you stop.
5. The national body is the root. The university is the intermediate. The degree is the leaf.
6. Now the important part: the national body keeps its own official seal in a locked vault and almost never takes it out. It delegates day-to-day signing to the universities. Where this comparison breaks: you can choose not to accept a degree. Software cannot exercise judgement, so the list of accepted bodies must be exactly right. Also, a national body has one country. A root certificate authority is trusted for every name on the internet, which is a very strange amount of power for a private company to hold.

32.6.3 PLAIN – a worked example

1. Here is a real chain, captured live in this sandbox. Read the `s:` line as "subject" and the `i:` line as "issuer".

```
Certificate chain
 0 s:CN = github.com
   i:O = Anthropic,
     CN = Egress Gateway SDS Issuing CA (production)
   a:PKEY: rsaEncryption, 2048 (bit); sigalg: RSA-SHA256
   v:NotBefore: Aug  6 23:49:45 2026 GMT;
     NotAfter: Sep  5 23:50:45 2026 GMT
 1 s:O = Anthropic,
     CN = Egress Gateway SDS Issuing CA (production)
   i:O = Anthropic,
     CN = sandbox-egress-gateway-production Egress Gateway CA
 2 s:O = Anthropic,
     CN = sandbox-egress-gateway-production Egress Gateway CA
   i:O = Anthropic,
     CN = sandbox-egress-gateway-production Egress Gateway CA
```

2. Read it as a ladder. Certificate 0 is the leaf for `github.com`. Its issuer is the subject of certificate 1. Certificate 1's issuer is the subject of certificate 2. Certificate 2 is its own issuer, so it is a root.
3. That is exactly the shape you want: leaf, intermediate, root.
4. What is wrong with it is who those names are. That is section 32.9.
5. On the reader's own Mac at home, running the same command against `github.com`, positions 1 and 2 would carry the name of a public certificate authority instead, and position 2 would match a certificate already in the system trust store.
6. Here is the trust store on this machine, which is a Debian-family Linux:

```
$ ls /etc/ssl/certs/*.pem | wc -l
152
$ grep -c "BEGIN CERTIFICATE" \
    /etc/ssl/certs/ca-certificates.crt
152
```

7. And here are four real roots from it, with their real dates:

Root	Not before	Not after
GlobalSign Root CA	Sep 1998	Jan 2028
Baltimore CyberTrust	May 2000	May 2025
DigiCert Global Root	Nov 2006	Nov 2031
ISRG Root X1	Jun 2015	Jun 2035

32.6.4 PLAIN - what is really happening inside

1. Why intermediates exist at all, in four reasons.
2. **Protection.** The root private key can be kept offline, in a hardware module in a safe, powered up a few times a year under camera. The intermediate key is online and doing daily work. If the online one is stolen, you replace the intermediate, not the root.
3. **Recovery.** You cannot fix a compromised root. It is baked into a billion devices and updating them takes years. You can fix a compromised intermediate in days.
4. **Constraint.** An intermediate can be limited by policy or, in some cases, by a name constraint, so it can only issue for certain domains.
5. Now path building. The browser has a leaf and possibly some extra certificates. It must find a route to a trusted root.
6. It looks at the leaf's issuer name and authority key identifier, and searches: first the certificates the server sent, then its own store, then any cached intermediates it has collected before.
7. This is a search, not a straight line. There can be several valid paths. A single intermediate is often cross-signed by two different roots so it can be validated by both old and new devices.
8. Once a candidate path is found, it validates every link: signature correct, dates valid, `CA:TRUE` present on every non-leaf, `keyUsage` including certificate signing, path length not exceeded, and the final certificate present in the trust store.
9. A very common production bug: the server forgets to send the intermediate. It works in a browser, because browsers cache intermediates from earlier visits, and fails in `curl` or in a Java service, which do not. Always test with a fresh tool, not with your browser.

32.6.5 TECHNICAL - the engineer's version

1. Where the root store actually lives:

System	Location
macOS	System Roots keychain
Windows	Trusted Root cert store
Debian/Ubuntu	/etc/ssl/certs
Firefox	its own NSS store

2. On macOS, open Keychain Access and look at System Roots, or use `security find-certificate` and `security dump-trust-settings` from the terminal. The file behind it is `SystemRootCertificates.keychain` under `/System/Library/Keychains`.
3. On Windows, run `certlm.msc` for the machine store or `certmgr.msc` for the user store, and look under Trusted Root Certification Authorities. Windows can also fetch roots on demand from Microsoft Update.
4. On Debian and Ubuntu the source files live in `/usr/share/ca-certificates`, local additions go in `/usr/local/share/ca-certificates`, and `update-ca-certificates` rebuilds `/etc/ssl/certs/ca-certificates.crt`. On Red Hat family systems the directory is `/etc/pki/ca-trust/source/anchors` and the command is `update-ca-trust`.
5. Firefox has always shipped its own store, so a root added to the operating system does not automatically work in Firefox. There is an enterprise policy to make Firefox also read the system store.

6. Chrome historically used the operating system store. It began shipping its own Chrome Root Store in 2022, from Chrome 105 onwards, becoming the default on Windows and macOS later that year. Treat the exact version numbers as approximate; the direction is not in doubt.
7. Who decides what goes in: the four root programmes are Mozilla, Microsoft, Apple and Google. Each publishes its own policy. The CA/Browser Forum, founded in 2005, publishes the Baseline Requirements that the programmes reference.
8. To get thrown out takes much less. Real examples: DigiNotar removed in September 2011 after a breach, and bankrupt within the month. WoSign and StartCom distrusted through 2016 and 2017 for backdating certificates and misissuance. Symantec's entire authority business distrusted by Chrome in stages through 2018, with the business sold to DigiCert in 2017.
9. What a certificate authority actually verifies before issuing, in three levels.
10. **Domain Validation (DV).** Proves only that the applicant controls the domain right now. Methods, from the Baseline Requirements: put a specific token at a specific path over HTTP, publish a specific DNS TXT record, respond to email at `admin@` or the address in WHOIS, or answer a TLS-ALPN challenge. Time taken: seconds, fully automated.
11. **Organisation Validation (OV).** Adds a check that a legal entity of that name exists, usually against a government register and a phone directory. Time taken: hours to days.
12. **Extended Validation (EV).** Adds documents, a lawyer's or accountant's confirmation, and a callback. Guidelines published by the CA/Browser Forum in 2007. Time taken: days to weeks.
13. How weak is this really? Domain Validation, which is what almost every site uses, proves control of DNS or of a web path. Anyone who can hijack DNS or BGP for a few minutes can pass it. It says nothing about identity.
14. The blunt summary: the modern web's certificate system proves control of a name. It stopped trying to prove who anybody is, because that turned out not to work.
15. Certificate Transparency, RFC 6962 of June 2013 and RFC 9162 of December 2021, is the real check today. Every publicly trusted certificate must be logged to public append-only logs, and Chrome refuses certificates without proof of logging. Domain owners can therefore see certificates issued for their names by anyone.

32.6.6 WORDS - remember these

1. Leaf — the website's own certificate — the end-entity certificate with `CA:FALSE` and the site's names in its SAN list.
2. Intermediate — the middle link — a subordinate CA certificate signed by a root and used for day-to-day issuance.
3. Root — the anchor of trust — a self-signed CA certificate present in the verifier's trust store, kept offline where possible.
4. Trust store — your machine's list of accepted anchors — an operating system or application collection of root certificates.
5. Path building — finding a route to a trusted anchor — the search through supplied, stored and cached certificates for a valid chain.
6. Certificate Transparency — public logging of every certificate — append- only logs, mandatory for Chrome, defined in RFC 6962 and RFC 9162.

32.7 What the browser verifies, and what it does NOT

32.7.1 PLAIN - in simple words

1. When the padlock appears, the browser has checked a short and very specific list.
2. It checked the signature chain reaches a root it trusts.
3. It checked today's date sits between the two dates in the certificate.
4. It checked the hostname you typed appears in the certificate's list of names.

5. It checked that no certificate in the chain claims permissions it should not have.
6. It checked revocation, badly, and often not at all.
7. That is the entire list. Now the other list.
8. It did **not** check that the site is honest.
9. It did **not** check that the company behind it is real, registered, or solvent.
10. It did **not** check that the content is safe, that the download is not malware, or that the form is not a phishing page.
11. It did **not** check that the name is the one you meant. A certificate for `paypa1.com` is perfectly valid; it just is not PayPal.
12. The padlock has never meant "this site is safe". It means "the pipe to whoever owns this name is private". Those are completely different claims.

32.7.2 PLAIN - a picture in your head

1. Think of a sealed pneumatic tube between your desk and an office somewhere.
2. The padlock is a guarantee about the tube. Nobody can read what goes through it. Nobody can slip a different note in. The tube really does end at the office with that name on the door.
3. It is not a guarantee about the office. The people in that office may be thieves. The name on the door may have been chosen last night to look like a bank.
4. A criminal can rent an office, put any name on the door, and order a perfectly good sealed tube from the tube company. The tube company only checks that the criminal really does control that door.
5. And that is exactly what happens: phishing sites almost all use HTTPS now, because certificates are free and automated.

Where this comparison breaks: a physical office costs money and can be raided. A domain costs a few dollars, is registered in minutes, and can be abandoned the moment it is reported. The economics are entirely different, which is why the "green padlock means safe" idea was retired.

32.7.3 PLAIN - a worked example

1. Take a real user story. Someone receives a message about a parcel and opens a link.
2. The address bar shows a padlock. There is no warning. The certificate is valid, issued minutes earlier by a free automated authority.
3. Every browser check passes: chain to a trusted root, dates fine, hostname matches the SAN entry exactly, `CA:FALSE`, correct key usage.
4. The page is a copy of a courier company's site, asking for a card number.
5. TLS did its job perfectly. It delivered the card number, encrypted and untampered, straight to the criminal.
6. Now contrast that with what the browser actually would have blocked: an attacker sitting on the cafe wireless trying to intercept the real courier site. That attacker cannot get a certificate for the real name, so the browser stops him.
7. Two different threats. TLS solves one of them completely and the other one not at all.
8. Modern browsers do have a defence against the phishing case, but it is a separate system: Google Safe Browsing, Microsoft SmartScreen and similar reputation lists. That is content safety, not transport security, and it works on a blocklist rather than on mathematics.

32.7.4 PLAIN - what is really happening inside

1. The hostname check is more subtle than people think.
2. The browser takes the name from the URL, not from DNS. If you typed `github.com`, the answer must contain `github.com`, whatever address the connection went to.
3. It compares against the SAN list only. The Common Name is ignored by modern clients.
4. Wildcards match one label only, and only at the leftmost position. `*.example.com` matches `api.example.com` but not `example.com` itself and not `a.b.example.com`.

5. The date check uses the client's own clock. A device with a badly wrong clock reports every certificate as expired or not yet valid. This is a very common cause of "all HTTPS is broken" on cheap devices and on machines whose battery died.

32.7.5 TECHNICAL - the engineer's version

1. The full check list a conforming client performs, per RFC 5280 and RFC 9525:

Check	Enforced
Chain to trusted root	strictly
Signature on each link	strictly
notBefore / notAfter	strictly
Hostname against SAN	strictly
basicConstraints CA	strictly
keyUsage, extKeyUsage	strictly
Name constraints	strictly
Certificate Transparency	Chrome: yes
Revocation	weakly or not

2. **CRL**, Certificate Revocation List. The authority publishes a signed list of revoked serial numbers. Problem: the list grows without bound, downloading it is slow, and it may be hours or days out of date.
3. **OCSP**, Online Certificate Status Protocol, RFC 6960 of June 2013. The client asks the authority about one specific serial number. Problems: it adds a network round trip to a third party during page load; it tells the authority which sites you visit, which is a privacy leak; and if the responder is unreachable, clients "soft-fail" and continue anyway.
4. Soft-fail is the fatal flaw. An attacker who can intercept your connection can also block your OCSP query, and then the check silently passes. A check an attacker can turn off is not a check.
5. **OCSP stapling**, the `status_request` extension from RFC 6066. The server fetches its own signed OCSP response periodically and staples it into the handshake. This fixes the privacy leak and the extra round trip. It does not fix soft-fail, because a server can simply not staple.
6. In 2024 and 2025 Let's Encrypt withdrew from OCSP entirely for privacy and cost reasons. It announced the plan in December 2024, dropped OCSP URLs from certificates on 7 May 2025, and switched off its OCSP responders on 6 August 2025. It relies on CRLs and on short lifetimes instead.
7. **Short-lived certificates are the real direction of travel.** If a certificate only lives days, revocation barely matters, because expiry does the job.
8. Let's Encrypt issued its first six-day certificate on 19 February 2025. Combined with the CA/Browser Forum schedule that takes the maximum to 47 days in March 2029, the industry answer to broken revocation is simply to make certificates expire faster than problems can spread.
9. Where experts disagree: some argue hard-fail revocation should have been mandated from the start and that soft-fail was a failure of nerve. Others argue hard-fail would have made the web fragile and dependent on authority uptime, and that short lifetimes are the better engineering answer. Both positions are held by serious people.

32.7.6 WORDS - remember these

1. Padlock — the browser's "connection is private" mark — an indicator that chain, dates, hostname and constraints validated; nothing more.
2. Hostname verification — the name you typed is in the certificate — a match against `subjectAltName` entries per RFC 9525.
3. Revocation — cancelling a certificate before expiry — publishing its serial as invalid via CRL or OCSP.
4. OCSP — asking the authority about one certificate — an online status query, RFC 6960, usually soft-failing and now being retired.

5. Stapling — the server carries its own status proof — the `status_request` TLS extension, removing the client's third-party lookup.
6. Soft-fail — carrying on when the check does not answer — the default behaviour that makes revocation checking defeatable by an attacker.

32.8 Self-signed certificates and the reader's own router

32.8.1 PLAIN - in simple words

1. A **self-signed** certificate is one where the subject and the issuer are the same. The key vouches for itself.
2. Nobody else has checked anything. There is no chain. There is nothing to follow upward.
3. So the browser cannot answer the question "does this reach a root I trust", and it stops and warns you.
4. The warning is not saying the encryption is weak. The encryption is exactly as strong as any other TLS connection.
5. The warning is saying: I cannot tell you who is at the other end.
6. That is the honest message. Everything else follows from it.
7. The reader's home router lives at `192.168.0.1`. Opening it in a browser over HTTPS produces exactly this warning, on every home router ever made.

32.8.2 PLAIN - a picture in your head

1. Somebody hands you a business card that says "I am the manager".
2. The card was printed by that same person. There is no company logo, no reference you can call.
3. If you are standing inside your own house and the person handing it over is the box on your own shelf that you plugged in yourself, the card tells you nothing you did not already know, and you do not need it.
4. If you are on a street in another city and a stranger hands you the same self-printed card claiming to be your bank manager, the card is worthless and the situation is alarming.
5. Same card. Completely different meaning, because of where you are standing and what else you know.

Where this comparison breaks: you can see a person and a shelf. You cannot see a network path. The reason the router case is safe is not that it feels close, but that the path is provably short, which the next block explains.

32.8.3 PLAIN - a worked example

1. Take the reader's real setup. The default gateway is `192.168.0.1`. The Mac is on the same wireless link as that box.
2. The certificate the router presents is self-signed. Here is the shape of one, generated for this chapter:

```
Subject: CN = router.home
Issuer : CN = KedByte Home Root CA
SAN    : DNS:router.home, IP Address:192.168.0.1
```

3. Now ask the only question that matters: could anything be sitting between the Mac and the router, pretending to be the router?
4. To do that, an attacker must already be on the reader's own wireless network or inside the flat with a cable. They would need the wireless password, or physical access.
5. But if an attacker is already on the local network with the wireless password, they can attack the router directly. Accepting the certificate warning does not make that worse.
6. So the risk added by clicking through, in this one situation, is close to zero. The endpoint is known, the path is one hop, and the attacker who could exploit it already has better options.
7. Now change one thing. The reader is on cafe wireless, types a bank address, and the same warning appears.

8. Here the path is long, the endpoint is unknown, and the warning is consistent with exactly one thing: somebody on the path is terminating the connection and cannot produce a real certificate.
9. That is the definition of an active interception attempt in progress. Clicking through hands them everything.
10. The asymmetry in one line: on the router you already know the endpoint, so the certificate adds nothing. On a public site the certificate is the only thing that identifies the endpoint, so ignoring it removes everything.

32.8.4 PLAIN - what is really happening inside

1. Clicking "proceed anyway" does not weaken the encryption. The session is still AES-GCM with an ephemeral key exchange.
2. What it removes is the authenticity property from section 32.2. You keep two of the three guarantees and throw away the one that made them useful against an active attacker.
3. Doing it properly instead has two options, and both remove the warning without removing the guarantee.
4. **Option one: trust that one certificate.** Export the router's certificate once, check its fingerprint over a channel you trust, and add it to your machine's trust store as an anchor.
5. **Option two: run a tiny private authority.** Create your own root once, add that root to your machines, and issue certificates for every device on your network from it.
6. Option two scales. Option one does not, but for one router it is fine.
7. Either way you have made a deliberate, narrow decision, instead of teaching yourself to click through warnings, which is the real long-term harm.

32.8.5 TECHNICAL - the engineer's version

1. A self-signed certificate for an IP address must carry that address as an `IP Address` entry in the SAN extension. A DNS entry with the same digits does not match, and modern clients will reject it.
2. Adding a private root, per platform:

System	Command or place
macOS	security add-trusted-cert
Debian	/usr/local/share/ca-certificates
RHEL	/etc/pki/ca-trust/source/anchors
Firefox	Settings, View Certificates

3. Understand what you have done: a root you add is trusted for **every name on the internet**, not just your router. A private root whose key is on a laptop that gets stolen is a total compromise of that laptop's browsing. Use `nameConstraints` if your tooling supports it.
4. **Certificate pinning** narrows trust in the other direction. The application refuses any certificate except a specific key it was built to expect, ignoring the trust store entirely.
5. On the web, pinning was standardized as HTTP Public Key Pinning, RFC 7469 of April 2015. It was a failure: a wrong pin bricked your own site for months with no recovery. Chrome removed support in version 72, early 2019. Do not use it.
6. **HSTS**, HTTP Strict Transport Security, RFC 6797 of November 2012, is the mechanism that survived. A response header tells the browser to use HTTPS for this host for a period, and to refuse to let the user click through certificate warnings at all.
7. Here is a real one, from the `github.com` response captured earlier in this chapter:

```
strict-transport-security: max-age=31536000;
  includeSubDomains; preload
```

8. `max-age=31536000` is one year in seconds. `includeSubDomains` extends it to every subdomain. `preload` is a request to be added to a list that ships inside the browser itself.
9. The preload list matters because HSTS normally needs one successful HTTPS visit before it can protect you. Preloading closes that first-visit gap by baking the rule into the browser binary.
10. HSTS is also why you cannot click through a warning on many major sites, and that is deliberate.

32.8.6 WORDS - remember these

1. Self-signed — the key vouches for itself — subject equals issuer, no chain to any external anchor.
2. Trust anchor — a certificate you accept without proof — a root installed in a trust store, trusted for all names unless constrained.
3. Private CA — your own small authority — a locally generated root used to issue certificates for internal hosts.
4. Pinning — accept only one specific key — an application-level restriction that bypasses the trust store entirely.
5. HSTS — always use HTTPS for this host — the `Strict-Transport-Security` header from RFC 6797, which also blocks warning bypass.
6. Preload — the rule ships inside the browser — inclusion in the browser's built-in HSTS list, protecting the very first visit.

32.9 When TLS is deliberately intercepted

32.9.1 PLAIN - in simple words

1. Everything in this chapter says a man in the middle cannot read your traffic, because he cannot produce a trusted certificate.
2. There is one way around that: put a certificate authority of your own into the machine's trust store.
3. Once that root is trusted, the middlebox can mint a certificate for any site on demand, and the browser will accept it happily.
4. This is not a bug. It is the system working exactly as designed. Trust stores are lists, and whoever controls the list controls the trust.
5. Employers do this on company laptops. Antivirus products do it on home machines. Developers do it on purpose with debugging tools.

32.9.2 PLAIN - a picture in your head

1. Imagine your office post room is instructed to open every sealed envelope, read it, photocopy it, put the contents in a fresh envelope, and reseal it with a company seal.
2. You were also given, on your first day, a stamp that says company seals are just as good as official ones.
3. So every letter you receive looks correctly sealed. You have no way to tell, from the seal alone, that it was opened, unless you look closely at whose crest is on it.
4. That closer look is exactly what the next block teaches.

Where this comparison breaks: a post room reads letters slowly. A TLS middlebox reads everything, all day, for thousands of people, and stores it. And unlike a post room, it can also modify content on the way through.

32.9.3 PLAIN - a worked example

1. This is real. The sandbox in which this chapter was written routes all outbound traffic through a company proxy.
2. Here is the actual command and the actual first lines of output when connecting to `github.com`:

```
openssl s_client -connect github.com:443 \
  -servername github.com </dev/null
```

```
depth=2 O = Anthropic,
  CN = sandbox-egress-gateway-production Egress Gateway CA
verify return:1
depth=1 O = Anthropic,
  CN = Egress Gateway SDS Issuing CA (production)
verify return:1
depth=0 CN = github.com
verify return:1
...
Verification: OK
Verify return code: 0 (ok)
```

3. Read it slowly. The subject at depth 0 is `github.com`. Correct.
4. The issuer, at depth 1, is not a public certificate authority. It is the proxy. The root at depth 2 is the proxy's own root.
5. And yet the last two lines say `OK`. Verification passed, because that root was installed into the machine's trust store on purpose.
6. Notice the validity dates on that leaf, from the same output: `NotBefore: Aug 6 23:49:45 2026` and `NotAfter: Sep 5 23:50:45 2026`. Exactly 30 days, minted by the box, not by GitHub.
7. If this were the reader's Mac at home, the issuer line would name a public authority instead. One line of output is the whole diagnosis.

32.9.4 PLAIN - what is really happening inside

1. The middlebox sits in the connection path. It sees the `ClientHello` and reads the server name extension.
2. It opens its own TLS connection outward to the real server, and validates that one properly, using the real public trust store.
3. It then generates a certificate for the requested name, signs it with its own private authority key, and completes a second, separate TLS handshake back towards you.
4. Two connections, joined in the middle, with the plaintext exposed at the join. That join is where scanning, logging or blocking happens.
5. Anything you type goes through that plaintext point: passwords, session cookies, private messages, source code.
6. Certificate pinning is the one thing that breaks this. An application that only accepts its own key refuses the forged certificate and fails loudly. That is why some mobile applications stop working on corporate networks.

32.9.5 TECHNICAL - the engineer's version

1. Categories of deliberate interception you will actually meet:

Kind	Who runs it	Typical issuer text
Corporate proxy	employer	company name, "SSL"
Antivirus scan	local software	product name
Debug proxy	you	mitmproxy, Charles
Captive portal	venue	portal or gateway name

2. Antivirus TLS scanning did the same thing on consumer machines. It has a poor security record; the 2017 paper "The Security Impact of HTTPS Interception" by Durumeric and others found many interceptors downgraded the outward connection to weaker cryptography than the browser would have used.
3. Debugging proxies are the honest version: `mitmproxy`, Charles Proxy, Fiddler and Burp Suite all install a local root and are indispensable for development. You install the root yourself and you should remove it when done.

4. Captive portals are not usually interception in this sense. They intercept plain HTTP and DNS to redirect you to a login page, and they cannot intercept HTTPS without a warning. That is exactly why an HTTPS-only device often fails to show the portal page at all.
5. How to detect it, in order of effort:

```
openssl s_client -connect github.com:443 \
  -servername github.com </dev/null 2>/dev/null \
  | openssl x509 -noout -issuer -dates
```

6. If the issuer is not a public authority you recognize, your traffic is being terminated somewhere. Compare with the same command from a phone on mobile data.
7. Certificate Transparency helps here too: an intercepting certificate is not logged publicly, whereas a genuine one must be. Chrome enforces this only for publicly trusted roots, and deliberately exempts locally installed roots, precisely so corporate interception keeps working.
8. Say this honestly: in many workplaces this is normal, disclosed in the acceptable use policy, and legal. It is also a genuine risk, because it concentrates every employee's plaintext at one appliance, and appliances get breached.

32.9.6 WORDS - remember these

1. TLS interception — somebody opens and re-seals your traffic — an on-path proxy terminating and re-originating TLS with a locally trusted root.
2. Secure web gateway — the corporate box doing it — an appliance performing inspection, filtering and data loss prevention.
3. Locally installed root — a trust anchor added by an administrator — a CA certificate placed in the machine store, exempt from Certificate Transparency enforcement.
4. mitmproxy — a developer's interception tool — an open-source intercepting proxy used for debugging and testing.
5. Captive portal — the login page on public wireless — a redirect of plain HTTP and DNS, which cannot transparently intercept HTTPS.

32.10 HTTP versions: 1.1, 2 and 3

32.10.1 PLAIN - in simple words

1. HTTP/1.0 opened a fresh TCP connection for every single file. A page with 40 images meant 40 connections.
2. HTTP/1.1 added **keep-alive**: reuse one connection for many requests. That was a huge saving.
3. But on one HTTP/1.1 connection the answers must come back in the order the questions were asked.
4. So one slow file blocks everything queued behind it. That is called **head-of-line blocking**.
5. HTTP/2 fixed it properly by chopping everything into small labelled pieces, so many exchanges share one connection and interleave.
6. But HTTP/2 still runs on TCP, and TCP itself insists on delivering bytes in order. So one lost packet still stalls every stream.
7. HTTP/3 fixed that by abandoning TCP and building a new transport on UDP, called QUIC, which keeps streams genuinely independent.

32.10.2 PLAIN - a picture in your head

1. HTTP/1.1 is a single-lane road. Cars arrive in order. One breakdown stops the lane.
2. HTTP/2 paints six lanes on that road. Cars can overtake. But the road is still one bridge, and if the bridge is blocked, all six lanes stop.
3. HTTP/3 replaces the bridge with six separate bridges. One collapsing affects only its own lane.

Where this comparison breaks: the HTTP/2 lanes are not physical; they are labels inside one byte stream. And the HTTP/3 bridges are still one UDP flow, sharing congestion control; only loss recovery is separated.

32.10.3 PLAIN - a worked example

1. A page needs one HTML file, one stylesheet and eight images: ten items.
2. Round trip time to the server is 100 milliseconds.
3. Under HTTP/1.1 with six connections and keep-alive: six handshakes, then ten requests spread over six lanes, roughly 400 milliseconds.
4. Under HTTP/2 on one connection: one handshake, all ten requests sent at once, answers interleaved, roughly 250 milliseconds.
5. Under HTTP/3: same as HTTP/2, plus setup folded into one round trip, plus no stall if a packet is lost, so the worst case improves most.

32.10.4 PLAIN - what is really happening inside

1. HTTP/2 is binary. Every message becomes **frames** with a type, a length and a stream identifier.
2. Because each frame carries its stream number, frames from different requests can be interleaved on one connection and reassembled at the far end. That is **multiplexing**.
3. Headers repeat enormously between requests on one page. HTTP/2 compresses them with HPACK, which keeps a shared table of previously seen headers so the second request sends an index instead of the text.
4. **Server push** let the server send a file you had not asked for, guessing you would need it. It sounded good and failed in practice: servers pushed things browsers already had cached, wasting bandwidth.
5. QUIC moves the reliability, ordering and encryption out of the kernel and into the application. It runs on UDP because UDP is the only transport that middleboxes will pass without inspecting.
6. QUIC merges the transport handshake and the TLS 1.3 handshake into one, so a new connection is one round trip, and a resumed one can be zero.
7. Each QUIC stream has its own delivery order. Losing a packet belonging to stream 3 does not hold up stream 7. That is the head-of-line fix.
8. **Connection migration**: a QUIC connection is identified by a connection ID, not by the four-part address pair. Move from wireless to mobile data and the connection survives with a different IP address.
9. It had to live in user space because changing TCP means changing every operating system kernel and every middlebox on earth, which takes decades. Shipping a library inside a browser takes weeks.

32.10.5 TECHNICAL - the engineer's version

Version	Year	Transport	Framing
HTTP/1.0	1996	TCP	text
HTTP/1.1	1997/1999	TCP	text
HTTP/2	2015	TCP + TLS	binary
HTTP/3	2022	QUIC on UDP	binary

1. Specifications: HTTP/1.0 is RFC 1945, May 1996. HTTP/1.1 was RFC 2068 January 1997 and RFC 2616 June 1999, now replaced by RFC 9112, June 2022.
2. HTTP/2 is RFC 7540, May 2015, now RFC 9113, June 2022. HPACK header compression is RFC 7541, May 2015.
3. QUIC is RFC 9000, May 2021, with TLS integration in RFC 9001 and loss recovery in RFC 9002. HTTP/3 is RFC 9114, June 2022, with QPACK header compression in RFC 9204.
4. HTTP/2 requires TLS in every browser implementation, though the specification permits cleartext. The ALPN token is `h2`; for HTTP/3 it is `h3`.

5. Server push was defined in RFC 7540 and disabled by default in Chrome 106 in 2022. Google reported that only about 1.25 percent of HTTP/2 sites used it, with no clear net performance gain. The replacement is the 103 Early Hints status code, RFC 8297, December 2017.
6. HTTP/3 is discovered, not requested. The server advertises it with an `Alt-Svc` response header or an HTTPS DNS record, and the client upgrades on a later connection.
7. Costs of QUIC: encryption and reassembly happen in user space, so CPU cost per byte is higher than kernel TCP, and some networks rate-limit or block UDP port 443 entirely. Clients therefore always keep a TCP fallback.

32.10.6 WORDS - remember these

1. Keep-alive — reuse one connection for many requests — persistent connections, default in HTTP/1.1.
2. Head-of-line blocking — one slow item stalls the rest — an in-order delivery constraint at the HTTP or TCP layer.
3. Multiplexing — many exchanges on one connection — independent streams identified by stream ID in HTTP/2 frames.
4. HPACK — squeeze repeated headers — the HTTP/2 header compression format, RFC 7541, using static and dynamic tables.
5. QUIC — a new transport on UDP — RFC 9000, providing streams, reliability and TLS 1.3 in user space.
6. Connection migration — keep the session when your address changes — QUIC's use of connection IDs instead of address pairs.

32.11 Cookies, sessions, and CORS

32.11.1 PLAIN - in simple words

1. HTTP has no memory. Each request stands alone. The server does not know the last request came from you.
2. That was deliberate, and it is why the web scaled. But logging in needs memory.
3. A **cookie** is the fix. The server sends a small piece of text. The browser stores it and sends it back on every later request to that site.
4. That is all a cookie is: a string the browser repeats back automatically.
5. A **session** is what the server does with it. Usually the cookie holds a random identifier and the server keeps the real data.
6. **Authentication** is proving who you are, once. **Session management** is staying recognized afterwards. They are different jobs and are attacked differently.

32.11.2 PLAIN - a picture in your head

1. You check into a hotel. At the desk you show your passport once. That is authentication.
2. They hand you a plastic key card with no name on it. That is the session cookie.
3. From then on, no one asks who you are. The card opens the door. That is session management.
4. Anyone who finds your card in the corridor can open your room, without ever seeing your passport.
5. The card stops working at checkout time, or if the desk cancels it. That is session expiry and revocation.

Where this comparison breaks: a card must be physically carried. A cookie is copied invisibly by any script that can read it, which is why `HttpOnly` exists to hide it from scripts entirely.

32.11.3 PLAIN - a worked example

1. Here are three real `Set-Cookie` headers, captured live from `github.com` in this chapter. The long session value has been shortened:

```
set-cookie: _gh_sess=Xkj3u...JWODdiSIORH1F; path=/;
  HttpOnly; secure; SameSite=Lax
```

```
set-cookie: _octo=GH1.1.1717559084.1786588114;
  expires=Fri, 13 Aug 2027 02:28:34 GMT;
  domain=.github.com; path=/; secure; SameSite=Lax
set-cookie: logged_in=no;
  expires=Fri, 13 Aug 2027 02:28:34 GMT;
  domain=.github.com; path=/; HttpOnly; secure;
  SameSite=Lax
```

2. `_gh_sess` has no `expires`, so it is a session cookie: it dies when the browser closes.
3. `_octo` has an expiry one year ahead, so it is persistent and survives restarts.
4. `domain=.github.com` widens it to every subdomain. Leaving `domain` out, as `_gh_sess` does, restricts it to exactly the host that set it, which is the safer default.
5. `path=/` means send it for every path on the site.
6. `secure` means never send this over plain HTTP.
7. `HttpOnly` means JavaScript cannot read it. That is what limits the damage of a cross-site scripting bug.
8. `SameSite=Lax` means do not send this cookie on cross-site requests except top-level navigations. That is the main defence against cross-site request forgery.
9. Note that `logged_in=no` is `HttpOnly` too. Even trivial state gets the protection, because it is free.

32.11.4 PLAIN - what is really happening inside

1. Mechanically: the server sends `Set-Cookie` in a response header. The browser stores name, value and attributes in a jar keyed by domain and path.
2. On every later request the browser looks up matching cookies and sends them in a single `Cookie` request header, without asking anyone.
3. **Session cookie versus token.** A session cookie is an opaque identifier; the server holds the state and can revoke instantly. A token, such as a signed JWT, carries the claims inside itself; the server needs no lookup but also cannot easily revoke it before expiry.
4. **Local storage** is a separate browser store that JavaScript reads and writes explicitly. It is never sent automatically, so it is immune to cross-site request forgery, and fully exposed to cross-site scripting because scripts can read it. Neither store is simply safer than the other.
5. Now the same-origin policy. An **origin** is the triple of scheme, host and port. `https://a.com` and `https://a.com:8443` are different origins. So are `http://a.com` and `https://a.com`.
6. By default a page may send a request to another origin, but its JavaScript may not read the response. That rule, from Netscape Navigator 2.0 in 1995, is what stops a random page reading your webmail.
7. **CORS** is the controlled exception. The receiving server opts in, using response headers, to let a named origin read the answer.
8. For anything beyond a simple request, the browser first sends a **preflight**: an `OPTIONS` request asking whether the real one is allowed.
9. This is the key point that confuses everyone: the server almost always processed the request perfectly. The browser then refused to hand the response to the page. So the error appears only in the browser console, and the server log shows a normal 200.
10. It follows that `curl` never sees a CORS error, and that adding headers on the client cannot fix one. Only the server can grant permission.

32.11.5 TECHNICAL - the engineer's version

1. Cookies were invented by Lou Montulli at Netscape in 1994. The current specification is RFC 6265, April 2011, with `SameSite` and other modern rules in the long-running RFC 6265bis work.
2. Attributes and what they do:

Attribute	Effect
Domain	widens to subdomains
Path	limits by URL prefix
Expires / Max-Age	persistent, not session
Secure	HTTPS only
HttpOnly	hidden from JavaScript
SameSite	cross-site sending rule

3. `SameSite` values: `Strict` never sends cross-site, `Lax` sends only on top-level GET navigations, `None` always sends and requires `Secure`. Chrome made `Lax` the default in Chrome 80, rolled out during 2020.
4. Here is a real preflight and its answer, captured from a small local server written for this chapter:

```
OPTIONS /api/profile HTTP/1.1
Host: api.kedbyte.test:8099
Origin: https://app.kedbyte.test
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: content-type

HTTP/1.1 204 No Content
Access-Control-Allow-Origin: https://app.kedbyte.test
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: Content-Type, Authorization
Access-Control-Max-Age: 600
Content-Length: 0
```

5. `Access-Control-Max-Age: 600` lets the browser cache that permission for ten minutes, so it does not preflight every call.
6. A request avoids preflight only if it is `GET`, `HEAD` or `POST`, uses no custom headers, and has a `Content-Type` of `application/x-www-form-urlencoded`, `multipart/form-data` or `text/plain`. Sending JSON triggers a preflight, which surprises everyone building an API.
7. To send cookies cross-origin you need `credentials: "include"` on the client and, on the server, `Access-Control-Allow-Credentials: true` plus an exact origin. The wildcard `*` is rejected with credentials, by design.

32.11.6 WORDS - remember these

1. Cookie — a string the browser repeats back — a name-value pair with attributes, stored per domain and path, RFC 6265.
2. Session — being recognized after login — server-side state referenced by an opaque identifier in a cookie.
3. HttpOnly — hidden from page scripts — a cookie attribute blocking `document.cookie` access, limiting XSS damage.
4. SameSite — the cross-site sending rule — `Strict`, `Lax` or `None`, the main defence against cross-site request forgery.
5. Origin — scheme plus host plus port — the security boundary of the same-origin policy.
6. Preflight — asking permission before the real request — an `OPTIONS` request with `Access-Control-Request-Method`.
7. CORS — a server opting in to cross-origin reads — response headers telling the browser to release the response to a named origin.

32.12 The whole journey, assembled

32.12.1 PLAIN - in simple words

1. This section answers the classic question in full: what happens when you type a web address and press Enter.
2. It uses the reader's own machine, the reader's own router at `192.168.0.1`, the reader's own resolver at `1.1.1.1`, and the address `github.com` resolved to `20.207.73.82`.
3. Everything in Part F of this book meets here in one list.
4. The reader's real session did not finish this journey. It stopped at a precise step, and the list below says exactly which one.

32.12.2 PLAIN - a picture in your head

1. Think of posting a letter to a company in another country.
2. You look up their address in a directory. That is DNS.
3. You walk to the post box, which is the only exit from your building. That is the default gateway.
4. The letter changes hands between postal services, none of which reads it. That is routing.
5. At the far end, a receptionist confirms the company's identity to you before you hand over anything valuable. That is the TLS handshake.

Where this comparison breaks: a letter goes once. A web page needs dozens of these exchanges, most of them to other companies entirely, and they overlap.

32.12.3 PLAIN - a worked example

1. The complete walkthrough. Times are realistic for the reader's connection.

```
key press -> DNS -> TCP -> TLS -> HTTP -> paint
```

2. Step 1. A key is pressed. The keyboard sends a scan code; the operating system turns it into a character and gives it to the browser.
3. Step 2. The browser's address bar decides: is this a search or a URL. A dot and no spaces means URL.
4. Step 3. The host name is normalized: lowercased, and if it contains non-ASCII it is converted to Punycode.
5. Step 4. The browser checks its HSTS list. `github.com` is preloaded, so even `http://` would be rewritten to `https://` before anything is sent.
6. Step 5. The browser checks its own HTTP cache. A fresh copy would end the story here with no network at all.
7. Step 6. The browser checks whether it already has an open connection to this host. Reusing one skips steps 8 to 26 entirely.
8. Step 7. Name resolution starts. The browser's own DNS cache is checked.
9. Step 8. Miss. The Mac sends a DNS query to the configured resolver, `1.1.1.1`, not to the router. This is the reader's real configuration.
10. Step 9. To send that packet the Mac needs a next hop. `1.1.1.1` is not on the local subnet, so the route table selects the default gateway, `192.168.0.1`.
11. Step 10. The Mac needs the gateway's hardware address, so it sends an ARP request for `192.168.0.1` and caches the reply.
12. Step 11. The resolver at `1.1.1.1` answers, from cache or by walking the root, then `.com`, then GitHub's own name servers.
13. Step 12. The answer arrives: `github.com` is `20.207.73.82`, with a time to live saying how long it may be cached.
14. Step 13. That address is in a Microsoft-owned range. GitHub has been owned by Microsoft since 2018 and fronts traffic through Microsoft's network edge.

15. Step 14. IPv6 was reported as none on this connection, so there is no happy-eyeballs race between families. IPv4 only.
16. Step 15. The browser opens a socket and starts a TCP connection to `20.207.73.82` port 443.
17. Step 16. The SYN goes to the gateway, `192.168.0.1`, which is hop 1 of the reader's traceroute.
18. Step 17. Hop 3, `137.97.29.249`, is a public address on the same ISP.
19. Step 18. Several hops showed two or three addresses. That is per-flow load balancing across parallel links, not an error.
20. Step 19. **This is where the reader's session stopped.** The SYN went out and nothing came back. No SYN-ACK, no reset, no ICMP unreachable. Silence, then a 15 second timeout in `curl`.
21. Step 20. What that proves: the connection never opened, so not one byte of TLS or HTTP was ever attempted. Steps 27 onwards did not happen on that day.
22. Step 21. In the normal case the server replies SYN-ACK and the client sends ACK. One round trip. The connection is open.
23. Step 22. TLS begins. The client sends `ClientHello` with its versions, random bytes, cipher suites, the `server_name` extension carrying `github.com`, ALPN offering `h2` and `http/1.1`, and a `key_share`.
24. Step 23. The server replies `ServerHello` with its chosen suite and its own `key_share`. Both sides now compute the shared secret.
25. Step 24. Everything after this point is encrypted, including the certificate.
26. Step 25. The server sends `EncryptedExtensions`, `Certificate`, `CertificateVerify` and `Finished`.
27. Step 26. The browser builds a chain from the leaf through the intermediate to a root in its trust store.
28. Step 27. It checks dates, `basicConstraints`, key usage, and that `github.com` appears in the certificate's SAN list.
29. Step 28. It checks Certificate Transparency proofs, and revocation, the latter weakly.
30. Step 29. It verifies `CertificateVerify`, which proves the server holds the private key right now.
31. Step 30. The client sends its own `Finished`. The handshake is complete in one round trip. Measured here, about 0.22 seconds.
32. Step 31. ALPN selected `h2`, so the browser speaks HTTP/2 and sends its request as compressed binary frames.
33. Step 32. The request may not reach GitHub's own servers at all. It may be answered by a load balancer or CDN edge that terminated TLS. Section 32.13.
34. Step 33. The response headers come back: `200 OK`, `content-type`, `strict-transport-security`, `set-cookie`, `content-security-policy`.
35. Step 34. The body streams in, usually compressed with gzip or brotli, and is decompressed on arrival.
36. Step 35. The HTML parser starts building the DOM tree while bytes are still arriving.
37. Step 36. It meets links to stylesheets, scripts, fonts and images, and issues those requests immediately over the same connection.
38. Step 37. DOM and CSSOM combine into a render tree of things that will actually be drawn.
39. Step 38. Layout computes the exact box and position of every element.
40. Step 39. Paint fills in pixels per layer, and compositing assembles the layers, often on the GPU.
41. Step 40. Remaining scripts run, fonts swap in, images decode, and the page fires its load event.
42. Step 41. The connection stays open for later requests, and a `NewSessionTicket` is stored so the next visit can resume faster.

32.12.4 PLAIN - what is really happening inside

1. Count the round trips in the good case: one for DNS, one for TCP, one for TLS 1.3. Three before the first HTTP byte.
2. The reader's failure sat in the middle of that list, at the TCP layer, and the shape of the failure was the clue.
3. A refused connection sends a reset. A dead server on a live network often produces an ICMP unreachable. A silent drop produces nothing.
4. Silence is what a firewall or filter configured to drop rather than reject produces, and it is also what a broken path produces.
5. What is proven: no response of any kind on that path, while the same request succeeded over mobile data from the same phone.
6. What is only suggested: where on the path the drop happened, and who caused it. The evidence does not name a party, and neither will we.

32.12.5 TECHNICAL - the engineer's version

1. The layered view, with the reader's real values.

```
Layer        Thing                Reader's value
-----------  -------------------  --------------------
Application  HTTP/2 request       GET / on github.com
Security     TLS 1.3 record       AES-GCM, X25519
Transport    TCP segment          dst port 443
Network      IPv4 packet          dst 20.207.73.82
Link         Ethernet / 802.11    next hop 192.168.0.1
```

2. Commands that observe each step, in order: `dig github.com`, `dig @1.1.1.1 github.com`, `netstat -rn` or `route -n get default`, `arp -a`, `traceroute github.com`, `nc -vz github.com 443`, `curl -v https://github.com`, `openssl s_client -connect github.com:443`, and the browser's network panel.
3. The single most useful diagnostic split: if `nc -vz host 443` succeeds but `curl` fails, the problem is above TCP. If `nc` itself hangs with no response, the problem is at or below TCP, which is exactly what happened here.

32.12.6 WORDS - remember these

1. Happy eyeballs — trying IPv6 and IPv4 together — RFC 8305, racing address families so a broken one costs milliseconds, not seconds.
2. Punycode — non-ASCII names in ASCII form — the encoding that turns international domain names into `xn--` labels.
3. Critical rendering path — the shortest route to first pixels — the DOM, CSSOM, render tree, layout, paint and composite sequence.
4. Round trip time — one message out and back — the quantity that dominates page load time on long paths.
5. Silent drop — a packet discarded with no reply — the failure mode that produces a timeout rather than an error, as in the reader's session.

32.13 CDNs, load balancers and caching in front

32.13.1 PLAIN - in simple words

1. The server you connect to is usually not the server that owns the site.
2. In front of it sit machines whose job is to be closer to you, to spread load, and to answer from memory when they can.

3. A **load balancer** takes one incoming connection and hands the work to one of many identical servers behind it.
4. A **CDN**, a content delivery network, is a large set of such machines placed in many cities, so most users reach one within a few milliseconds.
5. The real server behind all of it is called the **origin**.
6. This is why `github.com` answered from an edge site rather than from one room somewhere.

32.13.2 PLAIN - a picture in your head

1. A national newspaper does not post every copy from its head office.
2. It prints in regional plants near readers. Delivery is fast because the paper travelled a short distance.
3. The head office is the origin. The regional plants are the edge.
4. If a story changes after printing, every plant must be told to reprint. That is cache invalidation, and it is genuinely hard.

Where this comparison breaks: a newspaper is identical for everyone. Web responses vary by user, by language and by device, so the edge must know exactly which parts are shareable and which are personal.

32.13.3 PLAIN - a worked example

1. Real evidence from this chapter's live capture of `github.com`:

```
x-github-edge-region: iad
server: github.com
etag: W/"6a7d294b-8e2"
last-modified: Thu, 13 Aug 2026 02:17:47 GMT
cache-control: no-cache
```

2. `x-github-edge-region: iad` names the edge site, using the airport code for Washington Dulles. The answer came from an edge machine.
3. `etag` is a version tag for the content. On the next request the browser may send `If-None-Match` with that tag.
4. If the content has not changed, the server replies `304 Not Modified` with no body. That is a full round trip saved on bytes, though not on latency.
5. `cache-control: no-cache` does not mean "do not cache". It means "you may store it, but revalidate before every use". The header that forbids storage is `no-store`.

32.13.4 PLAIN - what is really happening inside

1. TLS is normally terminated at the edge. The edge holds a certificate and a private key for the site's name, decrypts your request, and then opens its own connection to the origin.
2. So the CDN sees your plaintext. That is unavoidable if it is going to cache or inspect anything.
3. That adds a party you must trust: your data is private from the network, and fully visible to the CDN operator.
4. Cache freshness has two mechanisms. A **time to live** says how long a copy may be used without asking. **Validation** asks the origin whether a stored copy is still good, using `ETag` or `Last-Modified`.
5. Invalidation is the hard part, because copies live in thousands of places. The industry answer is to avoid it: give every version of a file a unique name, cache it forever, and change the name when the content changes.
6. That is why built assets have hashed file names, and why the HTML that points at them is served with a very short cache time.

32.13.5 TECHNICAL - the engineer's version

1. Caching is specified in RFC 9111, June 2022. The directives that matter:

Directive	Meaning
max-age=N	fresh for N seconds
s-maxage=N	N seconds, shared caches
no-cache	store, but revalidate
no-store	do not store at all
private	browser only, not CDN
immutable	never revalidate
stale-while-revalidate	serve old, refresh

2. `Vary` tells caches which request headers change the answer. The real capture above showed `vary: Accept-Encoding, Accept, X-Requested-With`. A `Vary` on a high-cardinality header such as `User-Agent` destroys hit rates.
3. Layers you may meet in front of an origin, in order: DNS-based traffic steering or anycast, a network load balancer at layer 4, a reverse proxy or application load balancer at layer 7, a web application firewall, and the cache itself.
4. Anycast means the same IP address is announced from many locations, and routing delivers you to the nearest. It is why one address such as `1.1.1.1` answers from a different machine in every country.
5. The trade-off, stated plainly: a CDN reduces latency, absorbs attacks, and cuts origin cost. It also inserts a company that can read, modify, log and block every request, and whose outage becomes your outage.

32.13.6 WORDS - remember these

1. Origin — the real server behind everything — the authoritative source a cache or proxy fetches from on a miss.
2. Edge — a machine near the user — a point of presence terminating TLS and serving cached content.
3. TLS termination — where encryption ends — the point at which the request becomes plaintext, usually the edge, not the origin.
4. ETag — a version tag for content — an opaque validator compared with `If-None-Match` to produce a `304 Not Modified`.
5. Anycast — one address, many locations — the same prefix announced from many sites, with routing choosing the nearest.
6. Cache invalidation — telling every copy to forget — the hard problem usually avoided by giving each version a unique name.

32.98 Common wrong ideas

1. Wrong: the padlock means the site is safe. Right: it means the connection to whoever controls that name is private and untampered. Criminals get certificates too, in seconds, for free.
2. Wrong: HTTPS hides which site I visit. Right: the destination IP address is always visible, and until Encrypted Client Hello, RFC 9849 of March 2026, the server name was in the clear as well. DNS may leak it too.
3. Wrong: self-signed means insecure. Right: it means unverified. The encryption is identical. What is missing is any proof of who is at the other end, which matters enormously on a public site and hardly at all on your own router at `192.168.0.1`.
4. Wrong: encryption alone is enough. Right: without authentication you may have a perfectly encrypted tunnel to the attacker. Authenticity comes first; confidentiality is only useful afterwards.
5. Wrong: a certificate authority checks that the company is real. Right: almost all certificates are Domain Validated, which proves control of a name for a few seconds and nothing else.
6. Wrong: a CORS error means the server rejected my request. Right: the server usually processed it fine. The browser refused to give the response to your script. Only the server can grant permission.

7. Wrong: HTTP/2 solved head-of-line blocking. Right: it solved it at the HTTP layer, but TCP still delivers in order, so one lost packet still stalls every stream. HTTP/3 over QUIC fixed the transport layer.
8. Wrong: my company cannot read my HTTPS traffic. Right: if they installed a root certificate on the machine, they can read all of it, and one `openssl s_client` command shows you whether they do.

32.99 Chapter summary in 20 lines

1. HTTP is plain text: a request line, headers, a blank line, and a body. HTTPS is exactly that, carried inside TLS.
2. Methods differ by whether they are safe and whether they are idempotent, and those two properties drive caching, retries and 0-RTT rules.
3. Status codes come in five families; 4xx blames the request and 5xx blames the server, while 502 and 504 usually blame something in front of it.
4. TLS solves three problems: confidentiality, integrity and authenticity.
5. Encryption without authentication is nearly useless, because the tunnel may end at the attacker.
6. The building blocks are symmetric ciphers, hashes, message authentication codes, public key pairs and key exchange.
7. Diffie-Hellman lets two strangers agree a secret in public, using modular exponentiation that nobody can reverse at real key sizes.
8. Elliptic curves replaced RSA for new work because P-256 matches RSA 3072 in about a fifth of the bytes.
9. Ephemeral keys give forward secrecy, so a stolen server key cannot decrypt yesterday's recorded traffic.
10. TLS 1.2, RFC 5246 of 2008, needs two round trips. TLS 1.3, RFC 8446 of August 2018, needs one, by sending a key share in the first message.
11. 0-RTT saves another round trip and has no replay protection, so use it only for safe, idempotent requests.
12. A certificate binds a name to a public key and is trusted only because of who signed it, never because of what it says.
13. The chain runs leaf, intermediate, root; intermediates exist so the root key can stay offline and a compromise can be repaired.
14. Browsers check chain, dates, hostname against the SAN list, basic constraints and, in Chrome, Certificate Transparency.
15. Browsers do not check honesty, legitimacy, content safety, or whether the brand matches the name. The padlock never meant safe.
16. Revocation is largely broken; the industry answer is short-lived certificates, falling to a 47 day maximum from March 2029.
17. On your own router the endpoint is known and the link is one hop, so accepting a self-signed certificate once is a small risk. On a public site the same click may be surrendering to an active interception.
18. A locally installed root lets a proxy read everything. One `openssl s_client` command reveals it, by showing the issuer.
19. HTTP/2 multiplexed over one TCP connection; HTTP/3 over QUIC, RFC 9114 of June 2022, moved to UDP to remove head-of-line blocking in the transport.
20. Cookies give HTTP the memory it lacks; the same-origin policy and CORS decide who may read what, and a CORS error is always the browser's decision, never the server's.

33

CHAPTER

KB

The OSI and TCP/IP Models, Retrofitted to Your Own Session

33.0 What this chapter gives you

1. You will be able to say why networks are built in layers at all, what each layer is allowed to know, and what it is forbidden to know.
2. You will be able to name all seven OSI layers, define each one, and say what unit of data each one handles.
3. You will be able to explain that the OSI model came from a standards effort begun in 1977, published as ISO 7498 in 1984, and that the protocol suite it was built for lost to TCP/IP while the model survived as a teaching tool.
4. You will be able to name the four layers of the TCP/IP model, cite RFC 1122 from October 1989 as the document that states them, and map the two models against each other with real protocols in every row.
5. You will be able to place every single thing from the reader's own broken session at exactly one layer, and say which chapter covered it.
6. You will be able to walk one real HTTP request down the stack with real header sizes, and state what fraction of the bytes on the air were payload.
7. You will be able to show, with evidence, that layers 1, 2 and 3 worked in that session, that layer 4 did not complete, and that layers 5 to 7 were never reached at all.
8. You will be able to use the general fault rule: find the highest layer that works and the lowest that fails, and the fault is between them.
9. You will be able to list the places the model is simply wrong about reality, including TLS, VPNs, MPLS, NAT and QUIC, without losing faith in the model.
10. You will be able to spot the same layering pattern in graphics, storage and compilers, and see it as general engineering rather than a networking curiosity.

33.1 Why layers exist at all

33.1.1 PLAIN – in simple words

1. A network has to do many different jobs at once.
2. It must turn bits into a real signal, get data across one hop, get data across the world, notice losses and resend them, hide the contents from watchers, and agree what the contents mean.
3. So we split the work into **layers**. A layer is one level of the job.
4. Each layer does one kind of work. Each layer uses the layer below it and does not care how that layer works. Each layer offers a service upward and does not care what the layer above does with it.
5. The benefit is large. You can rip out one layer, put in a completely different one, and nothing above it needs to change.

6. The reader can carry the laptop out of the flat, join a friend's wired network instead of the home WiFi, and the same `git push` works.
7. And when a failure crosses layers, nobody owns it, because each layer can honestly say it did its own job correctly.

33.1.2 PLAIN - a picture in your head

1. Think about posting a birthday card to a cousin in another city.
2. You put it in an envelope and write the address. Now the card can be handled by people who will never read it.
3. A worker empties the post box into a sack with a label saying which sorting office it goes to. A van carries the sack down a road.
4. That is layering. Each level has its own wrapping, its own address format, and its own workers who ignore everything else.

Where this comparison breaks:

5. In the post, the envelope survives the whole journey. In a network, the outer wrapping is thrown away and rebuilt at every single router. Only the middle wrapping, the IP header, travels end to end.
6. The post does not chop your card into forty numbered pieces, send them by different roads, and glue them back together. TCP does exactly that.
7. And in the post the address on the envelope is the real address of the real house. On the internet, the address on the reader's packets was rewritten twice on the way out, by two different machines. That is NAT, Chapter 24.

33.1.3 PLAIN - a worked example

1. Here is the benefit, made concrete. The reader is on home WiFi in a flat in India, with a router at `192.168.0.1`. Now imagine plugging in a cable.

Layer of the job	On WiFi	On a cable
Radio or wire	5 GHz radio	copper pair
Link framing	802.11 frame	Ethernet frame
Link addresses	MAC address	MAC address
Internet address	192.168.0.x	192.168.0.x
Connection	TCP port 443	TCP port 443
Encryption	TLS 1.3	TLS 1.3
The request	git over HTTPS	git over HTTPS

2. The git program has no code in it for radio and none for copper. It does not know which is in use and cannot easily find out.
3. WiFi sometimes loses a frame because somebody switched on a microwave oven.
4. TCP was designed for wires, where "something is missing" almost always means the network is too busy. So TCP does the thing that helps a busy network. It slows down.
5. That is the price of the rule that a layer may not look below itself. It is also why WiFi does its own resending, so TCP never finds out.

33.1.4 PLAIN - what is really happening inside

1. A layer is not a place. It is a piece of code with a defined edge, and that edge is an **interface**: the short list of things the layer above is allowed to ask for.
2. The internet layer offers upward, roughly: "give me a lump of bytes and a destination address, and I will try to deliver it".

3. The transport layer offers a much stronger promise: "give me a stream of bytes and I will deliver them all, in order, exactly once".
4. Going down, each layer adds a small block of information at the front, called a **header**, holding only what that layer needs: addresses, lengths, numbers, flags.
5. So a packet is boxes inside boxes, with the message innermost. That wrapping is called **encapsulation**, and section 33.5 measures it in real bytes.
6. One strict rule: a layer must be replaceable without the layer above noticing. If replacing it breaks something above, the boundary was drawn in the wrong place.

33.1.5 TECHNICAL - the engineer's version

1. The OSI vocabulary is worth knowing because it still appears in specs. A layer N entity talks to its peer using a **protocol**, and to layer N-1 on the same machine using a **service**.
2. A PDU at one layer is an SDU at the layer below. That is the cleanest one-line definition of encapsulation there is.
3. The costs of layering are documented, not folklore. RFC 3439, "Some Internet Architectural Guidelines and Philosophy", by Randy Bush and David Meyer, December 2002, contains a section titled "Layering Considered Harmful".

Cost of layering	Concrete case
Duplicated work	802.11 and TCP both resend
Hidden state	TCP cannot see radio loss
Header overhead	128 bytes on a 190-byte GET
Fault ownership gaps	Every layer says "not me"

4. The counterweight is the end-to-end argument, from Saltzer, Reed and Clark, "End-to-End Arguments in System Design", presented in 1981 and published in ACM Transactions on Computer Systems in 1984.
5. Its claim: a function such as reliability can only be completely and correctly implemented at the end-points, so putting it in the network is at best a performance optimization, never a substitute.
6. Experts genuinely disagree about how strictly boundaries should hold. One camp treats any cross-layer visibility as technical debt. The other says wireless, mobility and congestion control cannot be done well without it.
7. Tools that isolate one layer on macOS: `ifconfig` for layers 1 and 2, `arp -a` for the layer 2 to 3 mapping, `netstat -rn` and `traceroute` for layer 3, `nc -vz` for layer 4, `openssl s_client` for TLS, and `curl -v` and `dig` for layer 7.

33.1.6 WORDS - remember these

1. Layer — one level of the job — a set of functions with a service interface upward and a peer protocol to the same layer remotely.
2. Header — the small label a layer adds — a prefix carrying that layer's control information only.
3. Encapsulation — a box inside a box — a layer N PDU becoming the payload, or SDU, of a layer N-1 PDU.
4. PDU — the parcel a layer sends — protocol data unit, the unit exchanged between peer entities at one layer.
5. End-to-end argument — the ends must check, the middle cannot be trusted to — the principle from Saltzer, Reed and Clark, 1981 and 1984.

33.2 The OSI seven-layer model

33.2.1 PLAIN - in simple words

1. In the late 1970s a large international committee set out to write down, once and for all, how computer networks should be built.
2. They split the job into seven layers, numbered 1 at the bottom to 7 at the top. Here they are, bottom first.
3. Layer 1, **physical**. Turning bits into something real: a voltage, a light pulse, a radio wave. It handles single bits.
4. Layer 2, **data link**. Getting a chunk of data across one hop, to the next box on the same local network. It handles frames.
5. Layer 3, **network**. Getting a chunk of data across many hops to a machine anywhere in the world. It handles packets.
6. Layer 4, **transport**. Making delivery reliable and ordered, and choosing which program on the machine gets it. It handles segments.
7. Layer 5, **session**. Starting, pausing and stopping a long conversation.
8. Layer 6, **presentation**. Agreeing how things are written down, and doing encrypting and squashing.
9. Layer 7, **application**. The thing you wanted: fetching a page, sending mail, pushing code.
10. Now the honest part. Layers 1, 2, 3 and 4 describe real things you can point at and measure.
11. Layers 5 and 6 barely describe anything that exists. In real systems that work is done inside the application or inside a library.

33.2.2 PLAIN - a picture in your head

1. Think of a large company that wrote a rulebook for how any two of its offices must exchange goods.
2. The rulebook has seven chapters. Chapter 1 covers the loading bay, 2 the local van, 3 the national route, 4 checking nothing was lost. Chapters 5, 6 and 7 cover the meeting: opening it, agreeing a language, and the business.
3. Then a smaller, scruffier company down the road ships goods using four scribbled pages, and wins every customer, because the four pages already worked while the seven chapters were still being argued over.
4. Twenty years later every logistics course still teaches the seven chapters, because they are the clearest description of the job. Everybody actually ships using the four scribbled pages.

Where this comparison breaks:

5. The OSI effort was not slow because the people were bad. It was slow because it was a genuine international consensus process across many governments and many computer makers, and consensus is slow.

33.2.3 PLAIN - a worked example

1. Place one ordinary thing at all seven layers: the reader's laptop asking `github.com` for something.
2. Layer 7: the intent, "give me the list of branches", expressed in HTTP.
3. Layer 6: the text is written as UTF-8 bytes and the whole thing is encrypted by TLS. Both are presentation-flavoured jobs.
4. Layer 5: the connection is opened, reused for several requests, then closed. In practice HTTP and TLS do this themselves. No separate session program exists.
5. Layer 4: TCP opens a connection to port 443, numbers every byte, and resends anything lost.
6. Layer 3: IP puts `20.207.73.82` on the front and asks the network to carry it.
7. Layer 2: 802.11 puts the router's hardware address on the front and sends it over WiFi to `192.168.0.1`.
8. Layer 1: the radio turns those bits into a 5 GHz signal in the air.
9. Notice how thin layers 5 and 6 were. That is not a flaw in the example. That is what the model looks like when you apply it honestly.

33.2.4 PLAIN - what is really happening inside

1. The unit of data has a different name at each layer, and engineers use the names precisely.
2. Bits at layer 1. Frames at layer 2. Packets at layer 3. Segments at layer 4 for TCP, datagrams for UDP. Just "data" at layers 5, 6 and 7.
3. Using the wrong word signals that you have not understood the model. A router does not forward frames end to end. It forwards packets and builds a new frame for each hop.
4. Layer 2 uses a MAC address, burned into the hardware, meaningful only on the local link.
5. Layer 3 uses an IP address, meaningful worldwide, changeable by software in a second.
6. Layer 4 uses a port number, which picks a program on one machine.
7. Four naming systems, and three lookups between them: DNS turns names into IP addresses, ARP turns IP addresses into MAC addresses, and the operating system turns port numbers into programs.

33.2.5 TECHNICAL - the engineer's version

1. The layers with their PDU names and real examples.

Layer	PDU	Real example
7 Application	data	HTTP, SMTP, DNS, SSH
6 Presentation	data	TLS, ASN.1 DER, JPEG
5 Session	data	RPC, NetBIOS, SIP
4 Transport	segment	TCP, UDP, SCTP, QUIC
3 Network	packet	IPv4, IPv6, ICMP
2 Data link	frame	Ethernet, 802.11, PPP
1 Physical	bit	1000BASE-T, 802.11ac

2. Layer 1, physical: the mechanical, electrical, functional and procedural means to activate and maintain a connection for bit transmission. It defines connectors, voltages, modulation and timing, and has no notion of a message.
3. Layer 2, data link: transfer across a single physical link, including framing, physical addressing, error detection and, on shared media, access control. It is bounded by the link. It does not route.
4. Layer 3, network: delivery across an arbitrary number of intermediate links, including logical addressing, routing and fragmentation.
5. Layer 4, transport: end-to-end delivery between processes, including multiplexing by port, reliability, ordering and flow control.
6. Layer 5, session: establishment, management and termination of dialogues, including checkpointing and synchronization points.
7. Layer 6, presentation: syntax and semantics of transferred data, including representation conversion, compression and encryption.
8. Layer 7, application: the interface to the application process, not the application itself.
9. The history, with dates. Work began in ISO Technical Committee 97, Subcommittee 16, in 1977, created specifically to produce an open systems architecture.
10. Charles Bachman, at Honeywell, is generally credited with proposing the seven-layer structure to that subcommittee in 1977 and 1978. He had won the ACM Turing Award in 1973 for his database work.
11. Hubert Zimmermann published the readable statement of it, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection", in IEEE Transactions on Communications, April 1980.
12. The model was published as the international standard ISO 7498 in 1984, and in parallel by CCITT, now ITU-T, as Recommendation X.200.
13. OSI was never only a model. It was a full protocol suite meant to replace everything: X.25 and CLNP at the network layer, TP0 to TP4 at transport, X.400 for mail, X.500 for directory, FTAM for files.

14. Governments mandated it. In the United States, FIPS 146, the Government OSI Profile or GOSIP, made OSI protocols a requirement for federal procurement from August 1990. The 1995 revision permitted TCP/IP and the mandate ended.
15. Why it lost is argued about, but the usual reasons are: TCP/IP was already deployed and free, the OSI specifications were large and expensive to obtain, implementations lagged the standards by years, and the ARPANET had completed its own switch to TCP/IP on 1 January 1983.
16. Andrew L. Russell's article "OSI: The Internet That Wasn't", IEEE Spectrum, July 2013, is the standard readable account.
17. Now the part most courses omit: OSI pieces that survived and that you use.

OSI-derived thing	Where you use it today
X.509 certificates	Every HTTPS connection
ASN.1 and DER	Certificate encoding, SNMP
IS-IS routing	Large ISP backbones
LDAP from X.500	Corporate directories

18. IS-IS deserves a pause. It is ISO/IEC 10589, an OSI routing protocol, adapted to carry IP by RFC 1195 in December 1990, and it runs inside some of the largest carrier backbones in the world.
19. X.509 came out of the X.500 directory work in 1988. Every TLS certificate the reader's browser checks is an X.509 certificate encoded in ASN.1 DER. Both are OSI presentation-layer artefacts.
20. On layers 5 and 6, be blunt. No mainstream internet protocol implements a separate session or presentation entity. RFC 1122 does not have them. Anyone who says TLS "is layer 6" is using a convenient label, not a standard.

33.2.6 WORDS - remember these

1. OSI — the seven-layer teaching model — Open Systems Interconnection Basic Reference Model, ISO 7498:1984 and ITU-T X.200.
2. Frame — the parcel on one hop — the layer 2 PDU, with link addresses and a frame check sequence.
3. Packet — the parcel for the whole journey — the layer 3 PDU, with source and destination IP addresses.
4. X.509 — the format of a website's certificate — the ITU-T certificate standard from the 1988 X.500 series, encoded in ASN.1 DER.
5. IS-IS — a routing protocol from the OSI family — ISO/IEC 10589, extended for IP by RFC 1195, widely used in carrier cores.
6. GOSIP — the government rule that OSI must be bought — US FIPS 146, mandatory from August 1990, relaxed in 1995.

33.3 The TCP/IP four-layer model

33.3.1 PLAIN - in simple words

1. The internet you are actually using was not built from the seven-layer model. It was built from a smaller, older, plainer set of ideas.
2. That set has four layers.
3. The **link** layer: getting bits across one hop. This is the OSI physical and data link layers rolled into one.
4. The **internet** layer: getting a packet across the world. This is IP, and only IP.
5. The **transport** layer: getting a stream to the right program, reliably. This is TCP and UDP.
6. The **application** layer: everything else. HTTP, DNS, SSH, mail, git. This is OSI layers 5, 6 and 7 rolled into one.
7. Notice the shape: a thin waist in the middle. There is exactly one internet layer protocol, IP, and everything above and below is plural.

8. That shape is called the hourglass. Many link technologies below, many applications above, one IP in the middle.

33.3.2 PLAIN - a picture in your head

1. Picture an hourglass made of glass, standing on a table.
2. The wide top is full of sand grains of every colour: web, mail, video calls, games, git, software updates, streaming.
3. The wide bottom is also full of different grains: WiFi, Ethernet, fibre, 4G, 5G, satellite, Bluetooth.
4. The narrow neck is one single grain type. That is IP. Every grain from the top must pass through it.
5. Because the neck is one thing everybody agreed on, the top and the bottom can each change completely without asking permission.

Where this comparison breaks:

6. The neck is not as narrow as the picture suggests. There are two IPs now, version 4 and version 6, and making them coexist has taken thirty years and is unfinished. The reader's machine reported `IPv6: (none)`, which is that unfinished job showing up in a flat in India.
7. And the neck is being widened from above. QUIC, in section 33.8, pushed transport logic up into applications, which the clean picture does not allow.

33.3.3 PLAIN - a worked example

1. Take the reader's `curl -v https://github.com` and label it in four layers instead of seven.

```
Application  HTTPS request for github.com, plus the
             TLS handshake, plus the earlier DNS query
Transport    TCP connection to 20.207.73.82 port 443
Internet     IPv4 packets, source 192.168.0.x,
             destination 20.207.73.82, TTL counting down
Link         802.11 frames over 5 GHz WiFi to the MAC
             address of 192.168.0.1
```

2. Try the same with seven layers and you will find yourself arguing about whether TLS is 5 or 6, and where the DNS query belongs.

33.3.4 PLAIN - what is really happening inside

1. The four-layer model is written into the document that tells you how to build a host: RFC 1122, October 1989, "Requirements for Internet Hosts - Communication Layers", edited by Robert Braden.
2. Its companion RFC 1123, published the same month, covers the application layer in detail. Together they are the closest thing the internet has to a constitution for end machines, and they are still cited in new work.
3. The reason the internet model came first is chronology. IP was specified in RFC 791 in September 1981 and TCP in RFC 793 the same month. ISO 7498 appeared in 1984. Working code beat the committee by three years.

33.3.5 TECHNICAL - the engineer's version

1. The mapping table, with real protocols in every row. This is the one to memorize.

OSI layer	TCP/IP layer	Real protocols
7 Application	Application	HTTP, DNS, SSH, SMTP
6 Presentation	Application	TLS, MIME, JPEG
5 Session	Application	HTTP keep-alive, RPC
4 Transport	Transport	TCP, UDP, QUIC, SCTP

OSI layer	TCP/IP layer	Real protocols
3 Network	Internet	IPv4, IPv6, ICMP, ARP
2 Data link	Link	Ethernet, 802.11, PPP
1 Physical	Link	1000BASE-T, 802.11ax

2. ARP is placed at "Internet" above, and that is a compromise. ARP has its own EtherType, `0x0806`, and rides directly on Ethernet, so it is not carried by IP, yet its entire purpose is to serve IP. RFC 1122 discusses it under the link layer. Textbooks place it at 2, at 3 and at "2.5". All are defensible.
3. ICMP is carried inside IP packets, protocol number 1, so structurally it sits above IP. Yet it is universally called a layer 3 protocol because its job is network control. RFC 1122 requires every host to implement it.
4. Numbers worth carrying for the protocols in that table:

Protocol	Document	Year
UDP	RFC 768	1980
IPv4	RFC 791	1981
TCP	RFC 793, now 9293	1981, 2022
ICMP	RFC 792	1981
ARP	RFC 826	1982
Host requirements	RFC 1122	1989
IPv6	RFC 8200	2017
TLS 1.3	RFC 8446	2018
QUIC	RFC 9000	2021
HTTP/1.1	RFC 9112	2022

5. RFC 9293, August 2022, is the current consolidated TCP specification. RFC 793 is obsolete as a citation, though every engineer still says "793" out of habit. RFC 8200, July 2017, is IPv6 as a full internet standard, replacing RFC 2460 from 1998.

33.3.6 WORDS - remember these

1. TCP/IP model — the four-layer picture of the real internet — the internet architecture as stated normatively in RFC 1122, October 1989.
2. Link layer — the "one hop" level — OSI layers 1 and 2 combined: framing, link addressing and the physical medium.
3. Internet layer — the "whole world" level — IP, plus ICMP and the routing protocols that populate the forwarding table.
4. Hourglass — many things above, many below, one in the middle — the property that IP is the single universal convergence layer.

33.4 The reader's own session, layer by layer

33.4.1 PLAIN - in simple words

1. Everything the reader saw in one broken evening is going to be put in its place in the model, one layer at a time.
2. That is worth doing for one reason: once a thing has a layer, a fault in it has a layer, and a fault with a layer is a fault you can find.
3. The setting: a flat in India, a home broadband line, a Mac on WiFi, and a `git push` to GitHub that would not go.

4. The evidence gathered was a traceroute that reached twelve hops, a `curl` that sat silent for fifteen seconds, a DNS answer that was correct, and the same site loading instantly on mobile data.

33.4.2 PLAIN - a picture in your head

1. Picture a tall office block with seven floors and a parcel chute down the middle.
2. On floor 7 somebody writes a note: "send me the list of branches". They fold it into a bag, label the bag, and drop it to floor 6.
3. Floor 5 labels which conversation it belongs to. Floor 4 adds a piece number so nothing is lost. Floor 3 adds the street address of the far building. Floor 2 names the next van driver. Floor 1 loads the van.
4. At the far building the same seven floors unwrap in the opposite order, each removing only its own label, until floor 7 reads the note.

Where this comparison breaks:

5. Nothing physically moves down the floors. The message is written once into one block of memory, and each layer writes its header into space reserved in front of it. There is no copying and no chute.
6. And the labels are not equal. The floor 3 label, the IP address, is the only one that survives the whole journey. The floor 2 label is torn off and rewritten at every building on the route.

33.4.3 PLAIN - a worked example

1. Here is the whole session sorted by layer. This table is the answer to "where does everything I have learned actually go".

Layer	From the reader's session	Where covered
1 Physical	5 GHz WiFi radio in air	Ch 23
1 Physical	Signal strength in dBm	Ch 23
2 Data link	802.11 frame and header	Ch 23
2 Data link	MAC address of router	Ch 23
2 Data link	ARP for 192.168.0.1	Ch 23
2 Data link	The switch in the box	Ch 23
3 Network	192.168.0.1 gateway	Ch 24
3 Network	20.207.73.82 target	Ch 24, Ch 25
3 Network	172.31.0.17, 172.26.x	Ch 24
3 Network	137.97.29.249 ISP edge	Ch 24
3 Network	TTL and time exceeded	Routing chapter
3 Network	BGP into ntwk.msn.net	Routing chapter
4 Transport	TCP SYN to port 443	TCP chapter
4 Transport	The 15-second silence	TCP chapter
5 and 6	TLS 1.3, never started	TLS chapter
7 Application	HTTPS and the HTTP GET	HTTP chapter
7 Application	DNS query to 1.1.1.1	Ch 25
7 Application	git pack and sideband	Git chapters

2. Eighteen rows, every one a real observed thing from one evening.

33.4.4 PLAIN - what is really happening inside

Layer 1, physical: the radio in the air.

1. What it is: the level that turns ones and zeros into something physical that travels. Voltage on copper, light in glass, or a radio wave in air.

2. What sits here: the WiFi radio inside the Mac and the matching radio inside the box at `192.168.0.1`, working at either 2.4 GHz or 5 GHz.
3. This layer is why walls matter and why a microwave oven can slow a download. Covered in Chapter 23, "Networking From Zero".

Layer 2, data link: getting across the room.

4. What it is: the level that moves a chunk of data across exactly one hop, to the next box sharing the same wire or the same air.
5. What sits here: the 802.11 frame, the addresses burned into the hardware, and the ARP question that found the router.
6. A MAC address is a six-byte number fixed to a network card. It means something only on the local network and never travels past the router.
7. The Mac knew the router's IP address, `192.168.0.1`, but a radio card cannot send to an IP address. It needs a MAC address.
8. So the Mac shouted to everybody on the local network: "who has 192.168.0.1, tell me your hardware address". That question is ARP.
9. The router answered, and the Mac then put that hardware address on the front of every frame it sent.
10. The box at `192.168.0.1` is three devices in one shell: a radio access point, a small switch, and a router. The switch part lives entirely at this layer. Covered in Chapter 23.

Layer 3, network: getting across the world.

11. What it is: the level that carries a packet across many hops, through machines that have never heard of you, to any address on earth.
12. What sits here: every single IP address the reader saw.
13. `192.168.0.1` is the router, a private address valid only inside the flat.
14. `20.207.73.82` is what `github.com` resolved to: a public address in a Microsoft-owned range.
15. `172.31.0.17` at hop 2, and the `172.26.x` addresses at hops 4, 5 and 6, are also private, but they belong to the reader's internet provider, not to the flat. A provider using private addresses in its own core is normal and is covered in Chapter 24.
16. `137.97.29.249` at hop 3 is a public address belonging to the provider.
17. Every IP packet carries a counter called TTL, time to live. Every router it passes reduces the counter by one. At zero the router discards the packet and sends back a complaint.
18. Traceroute abuses that on purpose. It sends a packet with the counter set to 1, then 2, then 3, and collects the complaints. Each one names a router. That is how the reader's twelve-hop list was made.
19. Routing is the question of which way to send a packet next. Between companies that is answered by BGP, the protocol by which every network tells its neighbours which addresses it can reach.
20. BGP is why hop 7 was the first machine named `ntwk.msn.net`. At that point the packets left the provider and entered Microsoft's own network.
21. Covered in Chapter 24 for addresses, and in the routing chapter for TTL, traceroute and BGP.

Layer 4, transport: reaching the right program, reliably.

22. What it is: the level that turns "a machine" into "a program on a machine", and turns unreliable delivery into a reliable stream.
23. What sits here: TCP, the number 443, and the fifteen seconds of silence.
24. An IP address gets you to a machine. A port number picks which program on it. 443 is the agreed number for secure web traffic.
25. TCP does not just start sending. It performs a three-step greeting.
26. Step one: the Mac sends a small packet marked SYN, meaning "I would like to open a connection".
27. Step two: the server should reply with SYN and ACK, meaning "yes, and I would like one too".

28. Step three: the Mac replies ACK, and only then may real data flow.
29. In the reader's session step one happened, over and over. Step two never happened, not once.
30. `curl` printed `Trying 20.207.73.82:443...`, waited fifteen seconds and gave up. That was the Mac patiently repeating step one. Covered in the TCP chapter.

Layers 5 and 6, the awkward ones: TLS.

31. What they are supposed to be: managing a conversation, and agreeing how things are written and encrypted.
32. What sits here: TLS, the thing that puts the S in HTTPS. It sits above TCP and below HTTP, turning a plain reliable stream into a secret, tamper-proof reliable stream.
33. The honest version: TLS does not fit. Its own specification never mentions an OSI layer number. It is not a session layer, because HTTP manages its own sessions. It is not a presentation layer, because it converts no data formats. It is wedged into a gap the model did not plan for.
34. In this session TLS never ran at all. TLS needs a working TCP connection before it can say its first word. Covered in the TLS chapter.

Layer 7, application: the thing you actually wanted.

35. What it is: the protocol that says what the request means.
36. What sits here: three separate application protocols, all involved in one `git push`.
37. HTTP is the request-and-response language of the web. HTTPS is that same language carried inside TLS.
38. DNS is the naming service. Before anything else, the Mac asked `1.1.1.1` what `github.com` meant and got back `20.207.73.82`.
39. Git has an application protocol of its own, spoken inside HTTP, which sends a pack of objects and receives progress messages back.
40. The error the reader saw twice, `send-pack: unexpected disconnect while reading sideband packet`, comes from that git protocol. The sideband is the side channel git uses for progress text while a pack transfers.
41. That is a layer 7 error message describing a layer 4 event. Section 33.6 returns to it, because it is the most useful thing in this chapter.
42. Covered in Chapter 25 for DNS, in the HTTP chapter for HTTP and HTTPS, and in the git chapters for the pack protocol.

33.4.5 TECHNICAL - the engineer's version

Layer 1, physical.

1. Home WiFi uses two delicensed bands: roughly 2.400 to 2.4835 GHz, and roughly 5.15 to 5.85 GHz. Exact permitted sub-bands differ by country and change over time, so check your national regulator rather than a textbook.
2. In the 2.4 GHz band only three 20 MHz channels do not overlap: 1, 6 and 11. In 5 GHz there are far more, bondable into 40, 80 or 160 MHz.
3. 802.11ac uses OFDM with up to 256-QAM, 8 bits per symbol; 802.11ax adds 1024-QAM at 10 bits per symbol. Subcarrier spacing is 312.5 kHz for 802.11a, n and ac.
4. Received signal strength in dBm is the number to look at.

RSSI	Practical meaning
-30 to -50 dBm	Excellent, next to the AP
-50 to -67 dBm	Good, video calls fine
-67 to -70 dBm	Usable, some retries

RSSI	Practical meaning
-70 to -80 dBm	Poor, rate drops hard
below -85 dBm	Effectively unusable

Layer 2, data link.

5. An 802.11 data frame header is 24 bytes: frame control 2, duration 2, three 6-byte address fields, and sequence control 2. A QoS data frame adds a 2-byte QoS control field, giving 26.
6. To carry IP, an 8-byte LLC/SNAP header is inserted, ending in EtherType `0x0800`. On Ethernet that EtherType sits in the 14-byte header instead.
7. WPA2 with CCMP adds an 8-byte header and an 8-byte message integrity check. A 4-byte CRC-32 frame check sequence closes every frame.
8. WiFi acknowledges every unicast frame at layer 2 and retransmits if the ACK is missing. Ethernet does not. That is the duplicated-reliability cost from section 33.1, made real.
9. ARP is RFC 826, November 1982, by David Plummer. A request is a broadcast to `ff:ff:ff:ff:ff:ff` with EtherType `0x0806`.
10. Inspect the cache with `arp -a`, which prints lines like `? (192.168.0.1) at 1c:3b:f3:0a:9e:71 on en0 ifscope [ethernet]`. An entry reading `(incomplete)` means the request went unanswered, which is a pure layer 2 fault. Watch it live with `sudo tcpdump -i en0 -n arp`.

Layer 3, network.

11. The IPv4 header is 20 bytes with no options: version and header length 1, DSCP and ECN 1, total length 2, identification 2, flags and fragment offset 2, TTL 1, protocol 1, header checksum 2, source 4, destination 4.
12. Address classification for the session, against RFC 1918, February 1996:

Address	Kind	Whose
192.168.0.1	Private, RFC 1918	The flat
172.31.0.17	Private, RFC 1918	The ISP core
172.26.22.235	Private, RFC 1918	The ISP core
172.16.18.33	Private, RFC 1918	The ISP core
137.97.29.249	Public	The ISP
104.44.196.187	Public	Microsoft
20.207.73.82	Public	Microsoft

13. The `172.16.0.0/12` block covers `172.16.0.0` through `172.31.255.255`, so every `172.16`, `172.26` and `172.31` address above is private. This is the range people most often misread.
14. TTL is an 8-bit field. macOS sets an initial TTL of 64. Each forwarding router decrements it and recomputes the header checksum, because the header changed.
15. At zero the router discards the packet and returns ICMP type 11 code 0, time exceeded in transit, per RFC 792, September 1981. Traceroute, written by Van Jacobson in 1987, exists entirely to harvest those.
16. macOS `traceroute` sends UDP probes to high ports by default, three per hop. `traceroute -I` uses ICMP echo, and `traceroute -P TCP -p 443` uses TCP, which is the more informative test when one port is suspected.
17. The reader's hops 5, 6, 8, 9 and 11 each printed more than one address. That is equal-cost multi-path forwarding: three probes took three parallel links. It is a healthy sign, not a fault.
18. From hop 7 every name ends in `ntwk.msn.net`, which is Microsoft network infrastructure. The city codes are airport-style: `del01` Delhi, `bom01` Mumbai, `pnq20` and `pnq21` Pune.

19. The role and interface codes are Microsoft naming conventions, not standards: `ibr` internal backbone router, `rwa` regional wide-area router, `owr` outer wide-area router, and `ae`, `be` and `po` all meaning a bundle of physical links acting as one logical link.
20. Inter-domain routing is BGP-4, RFC 4271, January 2006, descended from RFC 1105 in 1989. Microsoft's best known autonomous system number is 8075. I have not verified which AS announces the exact prefix containing `20.207.73.82`, so treat that attribution as likely rather than proven.

Layer 4, transport.

21. The TCP header is 20 bytes minimum. A macOS SYN typically carries 20 bytes of options, giving a 40-byte header and a 60-byte IP packet with no payload at all. Typical options are maximum segment size, window scale, selective acknowledgement permitted, and timestamps, with NOP bytes for alignment.
22. Port 443 is the IANA-registered port for HTTP over TLS. Port 22 is SSH. Both appear in this session and only one of them worked.
23. The initial retransmission timeout is 1 second per RFC 6298, June 2011, and doubles on each retry. So in fifteen seconds the Mac sends a SYN at roughly t=0, 1, 3, 7 and 15 seconds: about five packets, about 300 bytes, and zero bytes received.
24. macOS bounds the whole connect attempt with the sysctl `net.inet.tcp.keepinit`, default 75000 milliseconds, so 75 seconds. `curl` gave up at 15 seconds, well before the kernel would have.
25. What did not happen is the evidence. No TCP RST came back, which would have meant a closed port. No ICMP type 3 came back, which would have meant a device admitting it refused the packet. Nothing came back.
26. Confirm the shape with `sudo tcpdump -i en0 -n 'host 20.207.73.82 and tcp port 443'`. Repeated outbound `Flags [S]` lines with nothing inbound is the exact signature. Test one port quickly with `nc -vz -G 5 20.207.73.82 443`.

Layers 5 and 6, TLS.

27. The TLS record header is 5 bytes: content type 1, legacy version 2, length
    2. Under TLS 1.3 with an AEAD cipher such as AES-128-GCM, each record also carries a 16-byte authentication tag and one byte of real content type inside the encrypted body.
28. A full TLS 1.3 handshake costs one round trip before application data: ClientHello with a key share, then ServerHello, EncryptedExtensions, Certificate, CertificateVerify and Finished, then the client's Finished.
29. The ClientHello carries the server name extension, SNI, containing the text `github.com`. Whether that name is visible on the wire depends on whether Encrypted Client Hello is in use at both ends; check its current standards status before relying on it either way.
30. None of this ran. TLS is carried over TCP and TCP never opened, so the reader's machine never emitted a single TLS byte to `20.207.73.82`.
31. Test this layer alone with `openssl s_client -connect github.com:443 -servername github.com`, which separates a TLS fault from a TCP fault by showing how far it got.

Layer 7, application.

32. DNS is RFC 1034 and RFC 1035, November 1987, over UDP and TCP port 53. The reader's resolver was `1.1.1.1`, Cloudflare's public resolver, not the router. It answered `20.207.73.82`, correctly and fast.
33. HTTP/1.1 is now RFC 9112, June 2022, replacing RFC 7230 from 2014 and RFC 2616 from 1999. HTTP/2 is RFC 9113 and HTTP/3 is RFC 9114.
34. Git over HTTPS uses the smart HTTP protocol: a `GET` of `/info/refs?service=git-receive-pack` for discovery, then a `POST` to `/git-receive-pack` carrying the pack.

35. Git frames its own messages in pkt-line format: a four-character hexadecimal length prefix followed by that many bytes, with `0000` as a flush marker. That is layer 7 framing inside HTTP inside TLS inside TCP.
36. The sideband multiplexes three channels in one stream: channel 1 pack data, channel 2 progress text, channel 3 a fatal error. `send-pack: unexpected disconnect while reading sideband packet` means the stream ended while git was waiting for the next pkt-line on that side channel.
37. Observe layer 7 with `dig github.com`, `curl -v`, and `GIT_TRACE_PACKET=1 git push`, which prints every pkt-line git exchanges.

33.4.6 WORDS - remember these

1. MAC address — the hardware name of a card — a 48-bit link-layer address whose top 24 bits are an IEEE-assigned OUI.
2. ARP — the shout that finds a router's hardware address — Address Resolution Protocol, RFC 826, EtherType `0x0806`, broadcast request, unicast reply.
3. TTL — the counter that stops packets circling forever — an 8-bit IPv4 field decremented per hop, triggering ICMP type 11 at zero.
4. BGP — how networks tell each other what they can reach — Border Gateway Protocol version 4, RFC 4271, the inter-domain routing protocol.
5. SYN — the first word of a TCP conversation — the synchronize flag opening the three-way handshake, retransmitted with exponential backoff.
6. SNI — the site name sent before encryption starts — Server Name Indication, the TLS extension letting one address serve many certificates.
7. pkt-line — how git chops its own stream into messages — a four-hex-digit length prefix plus payload, with `0000` as a flush packet.
8. Sideband — the channel carrying progress text — git's multiplexing of pack data, progress and errors over one connection.

33.5 Encapsulation walked through with real numbers

33.5.1 PLAIN - in simple words

1. Encapsulation means putting a thing inside a bigger thing that carries it. Every layer does it, adding a small label at the front of what it was given.
2. So the message you cared about ends up buried in the middle of a much bigger bundle of bytes.
3. This costs you. The labels are real bytes, sent over the real air, taking real time, and you are paying for them.
4. The answer, for a small request over WiFi, is that roughly four bytes in every ten are labels rather than message.
5. For large transfers the picture is much better, because the labels stay the same size while the message gets bigger.

33.5.2 PLAIN - a picture in your head

1. Think of wooden nesting dolls, the kind where each one opens with a smaller one inside.
2. The smallest doll is a folded note: the thing you actually wanted to send.
3. Around it goes a doll marked "secret", which locks. That is TLS.
4. Around that, a doll marked "piece 4 of 9". That is TCP.
5. Around it goes a doll marked "secret", which locks. That is TLS.
6. Around that, a doll with the far building's street address. That is IP.
7. Around that, a doll naming the courier taking it to the end of the road. That is WiFi. You post the outermost doll.

Where this comparison breaks:

8. Real dolls get much thicker at each level. Network headers do not. The IP header is 20 bytes whether it wraps 1 byte or 1,460.
9. And a real doll survives the journey. The outermost network shell is stripped and rebuilt at every router, with different labels every time.

33.5.3 PLAIN – a worked example

1. Here is one real, complete HTTP request, of the kind git sends when it starts talking to GitHub.

```
GET /kedbyte/notes.git/info/refs?service=git-upload-pack HTTP/1.1
Host: github.com
User-Agent: git/2.39.5 (Apple Git-154)
Accept: */*
Accept-Encoding: deflate, gzip
Pragma: no-cache
```

2. Counting every character, including the carriage return and line feed ending each line and the blank line that closes the headers, that is exactly 190 bytes.
3. Now watch it grow on the way down, one layer at a time.

Step	Adds	Running total
HTTP request text	-	190 bytes
TLS 1.3 record	22 bytes	212 bytes
TCP segment	32 bytes	244 bytes
IPv4 packet	20 bytes	264 bytes
802.11 frame	54 bytes	318 bytes

4. Final answer: 318 bytes go over the air to carry 190 bytes of request.
5. That is 128 bytes of overhead, 40.3 percent of the frame. The payload is 59.7 percent.

33.5.4 PLAIN – what is really happening inside

1. Nothing is copied five times. The operating system allocates one buffer and deliberately leaves empty space at the front, called headroom.
2. Each layer writes its header backwards into that space and moves a pointer. The message bytes never move.
3. On macOS and the BSD family that buffer is an `mbuf`. On Linux it is an `sk_buff`. Both reserve headroom for exactly this reason.
4. This is why "the data travels down through the layers" is a teaching lie. Section 33.98 says so plainly.
5. The one place bytes really are copied is the boundary between the kernel and the network card, and modern cards use DMA to avoid even that.

33.5.5 TECHNICAL – the engineer's version

1. The full nesting, drawn to scale of description rather than of size.

```
+-----------------------------------------------------+
| 802.11 QoS data frame          318 bytes            |
| hdr 26|CCMP 8|LLC 8|...|MIC 8|FCS 4                 |
| +-------------------------------------------------+ |
| | IPv4 packet                   264 bytes         | |
| | IP header 20 bytes                              | |
| | +---------------------------------------------+ | |
| | | TCP segment                   244 bytes     | | |
| | | TCP header 20 + options 12                  | | |
| | | +-----------------------------------------+ | | |
```

```
| | | | TLS 1.3 record                 212 bytes   | | | |
| | | | rec hdr 5 + type 1 + AEAD tag 16           | | | |
| | | | +----------------------------------------+ | | | |
| | | | | HTTP GET request         190 bytes     | | | | |
| | | | | the only part anyone wanted            | | | | |
| | | | +----------------------------------------+ | | | |
| | | +--------------------------------------------+ | | |
| | +------------------------------------------------+ | |
| +----------------------------------------------------+ |
+------------------------------------------------------+
```

2. The 128 bytes of overhead, itemized so you can check every number.

Layer	Fields	Bytes
TLS 1.3	hdr 5, type 1, tag 16	22
TCP	header 20, options 12	32
IPv4	header, no options	20
802.11	QoS hdr 26, FCS 4	30
WPA2 CCMP	header 8, MIC 8	16
LLC/SNAP	for EtherType 0x0800	8

3. On wired Ethernet the layer 2 cost would be 18 bytes rather than 54: a 14-byte header and a 4-byte FCS. The same request would be 282 bytes on the wire and 67.4 percent payload.
4. Layer 1 on WiFi is not measured in bytes at all. It is measured in microseconds, and that is where the true overhead hides.
5. Approximate airtime for that single 318-byte frame on 5 GHz, 802.11ac, one spatial stream, 80 MHz, at a physical rate of 433 Mbit/s:

Component	Approximate time
DIFS wait	34 microseconds
Average backoff	67 microseconds
VHT preamble	40 microseconds
The 318 bytes	5.9 microseconds
SIFS plus ACK	60 microseconds
Total	about 207 microseconds

6. Read that again. The data itself occupies under 3 percent of the airtime. Everything else is protocol.
7. Efficiency improves enormously with size.

Payload	Frame on WiFi	Payload share
190 bytes	318 bytes	59.7 percent
536 bytes	664 bytes	80.7 percent
1,398 bytes	1,526 bytes	91.6 percent

8. 1,398 is not arbitrary. From a 1,500-byte MTU, subtract 20 for IP and 32 for TCP with timestamps to get an MSS of 1,448, then subtract 22 for the TLS record, leaving 1,426 bytes of application data. Round down further for a VPN or PPPoE header, which is why 1,398 and 1,400 are common settings.
9. Read real header sizes off the wire with `tcpdump -i en0 -v -X`, or open a capture in Wireshark, where each layer appears as an expandable tree in exactly the nesting shown above.

33.5.6 WORDS - remember these

1. Encapsulation — putting a message inside a bigger envelope — wrapping a layer N PDU as the payload of a layer N-1 PDU.
2. Overhead — the bytes that are not your message — header and trailer bytes as a fraction of the transmitted unit.
3. Headroom — the empty space in front of the message — reserved bytes at the start of a kernel packet buffer so headers can be prepended without copying.
4. MTU — the biggest frame a link will carry — maximum transmission unit, 1,500 bytes for standard Ethernet.
5. MSS — the biggest chunk of data one TCP segment carries — maximum segment size, the MTU minus the IP and TCP headers.
6. FCS — the check at the end of a frame — frame check sequence, a 32-bit CRC letting the receiver discard corrupted frames.

33.6 Where the reader's failure sits in the model

33.6.1 PLAIN - in simple words

1. The question is not "what is wrong". It is "which layer is wrong", because that is a question evidence can answer.
2. Layer 1 worked. The laptop was on the WiFi, frames were going out and coming back. If the radio were dead nothing at all would have happened.
3. Layer 3 worked, and this is the strong evidence. The traceroute reached twelve routers, the last inside Microsoft's network in Pune.
4. Think about what that proves. Packets from the flat crossed the provider's network, entered Microsoft's, travelled Delhi to Mumbai to Pune, and complaints from those routers came all the way back.
5. Layer 4 did not complete. The laptop said hello to port 443 about five times over fifteen seconds and was never answered.
6. Layers 5, 6 and 7 were never attempted. TLS cannot start without TCP, HTTP cannot start without TLS, git cannot start without HTTP.
7. So the fault is at layer 4, or in something that only affects layer 4 traffic to that address.
8. That is a far smaller question than "the internet is broken", and it was reached in about two minutes of evidence gathering.

33.6.2 PLAIN - a picture in your head

1. Imagine a lift in a tall building that is not arriving. You could stand there guessing, or you could check floors.
2. On floor 4 you press and nothing happens. No light, no sound, no error.
3. You now know something precise: the fault is at or just below floor 4, and floors 5, 6 and 7 have not been tested and cannot be, because you cannot get to them.

Where this comparison breaks:

4. In a building, a fault on floor 4 is on floor 4. On a network, the thing that broke layer 4 may sit at layer 3, in a router that drops only packets with certain port numbers.
5. So "the fault is at layer 4" means "the symptom is at layer 4". The cause may be a device operating below it that inspects above itself. Section 33.8 is about exactly that dishonesty.

33.6.3 PLAIN - a worked example

1. The evidence table. Left, the layer. Middle, what was observed. Right, what that proves.

Layer	Observation	Verdict
1 Physical	Associated, traffic flows	Works
2 Data link	Router reachable, ARP fine	Works
3 Network	Traceroute reached hop 12	Works
4 Transport	5 SYNs, 0 replies, 15 s	Fails
5 and 6 TLS	No ClientHello ever sent	Untested
7 Application	DNS answered correctly	Partly works
7 Application	HTTP never sent	Untested

2. That is not a contradiction. DNS to `1.1.1.1` is a different destination on UDP port 53, not TCP port 443. It took a different path with a different transport, and it succeeded.
3. The most useful thing was the mobile data test: the same request over a different provider's network succeeded instantly.
4. Change one variable, hold the rest. The variable that changed was the path from layer 1 to layer 3. The failure followed the path, not the machine and not the server.

33.6.4 PLAIN - what is really happening inside

1. **Find the highest layer that demonstrably works, and the lowest layer that demonstrably fails. The fault is between them.**
2. Work from the bottom up, because a low layer failing makes every test above it meaningless.
3. There is no point debugging a certificate error if ARP is failing. The certificate error is a lie told by a program that could not reach anything.
4. The traceroute proves that packets reached hop 12 and replies returned. It proves nothing about hops 13 onward, because routers commonly refuse to send the replies traceroute needs.
5. Silence at the end of a traceroute is normal and is not evidence of a fault. Saying otherwise is the most common mistake in network diagnosis.
6. What settled the reader's case was a TCP connection attempt that got no response of any kind, combined with the same attempt succeeding over a different network.
7. That combination supports one conclusion: something on that particular path silently discarded the packets. It does not identify who, and honest diagnosis stops there.

33.6.5 TECHNICAL - the engineer's version

1. The outcomes of a TCP connection attempt, and what each proves, are the sharpest diagnostic in networking.

Response	Meaning	Layer implicated
SYN-ACK	Port open, path fine	None, it works
RST	Reached host, port shut	7, service not running
ICMP type 3	A device refused it	3, policy in the path
Nothing at all	Silently discarded	3 or 4, on the path

2. The reader got the fourth row. A RST would have meant packets arrived and something answered. An ICMP unreachable would have meant a device admitted responsibility. Silence means neither.
3. The sequence of commands that produces this evidence, in order:

```
ifconfig en0 | grep -E 'status|inet '
arp -a | grep 192.168.0.1
ping -c 3 192.168.0.1
traceroute -n 20.207.73.82
traceroute -P TCP -p 443 -n 20.207.73.82
```

```
nc -vz -G 5 20.207.73.82 443
nc -vz -G 5 20.207.73.82 22
dig +short github.com @1.1.1.1
curl -v --max-time 15 https://github.com
```

4. Line 5 is the one most people omit and the most informative: a TCP traceroute to port 443 shows how far port 443 traffic specifically gets, which a UDP or ICMP traceroute cannot tell you.
5. Compare lines 6 and 7. In the reader's session one transport worked and the other did not, localizing the problem to something treating port 443 differently from port 22.
6. Note what that does not prove. It does not prove who is doing the treating, nor that it is deliberate. Consistent with the evidence: a filter, a misrouted prefix, an overloaded state table, or an asymmetric return path. Not consistent: the server being down, since the same address served the same content over mobile data.
7. The git error from the same session, `send-pack: unexpected disconnect while reading sideband packet`, is the same lesson in a different costume: a layer 7 message whose real cause was a layer 4 connection dying mid-stream.
8. That is why the error tells you nothing about whether the push landed. The disconnect happened while reading the response. The write may already have been applied on the server. The only correct action is to re-query the remote state, never to assume from the error text.

33.6.6 WORDS - remember these

1. Bottom-up diagnosis — check the ground floor first — testing layers in ascending order so no test is invalidated by a lower failure.
2. Silent drop — the packet vanishes with no complaint — a discard with no RST and no ICMP error, giving the sender no information at all.
3. RST — a slammed door — the TCP reset flag, meaning the host was reached and refused the connection.
4. ICMP unreachable — a written refusal — ICMP type 3, with codes for host, network, port and administrative prohibition.
5. Localizing — narrowing where the fault is — changing exactly one variable at a time until the failure follows the variable.

33.7 A fault-to-layer table

33.7.1 PLAIN - in simple words

1. If you learn the shape of each failure, the error message tells you where to look before you have run a single command.
2. The table in this section is the one to pin above your desk. Read it in either direction: from a symptom find the layer, or from a layer find the command that tests it.

33.7.2 PLAIN - a picture in your head

1. Think of a doctor with a list: fever means one family of causes, a rash another, a limp a third.

Where this comparison breaks:

2. A body has one owner. A network path crosses many, and the layer showing the symptom is often not the layer that owns the cause.

33.7.3 PLAIN - a worked example

1. Symptoms and the tool that settles each one.

Layer	What you see	Tool
1 Physical	No link, weak signal	ifconfig, RSSI
2 Data link	ARP incomplete	arp -a, tcpdump arp
3 Network local	Gateway unreachable	ping 192.168.0.1
3 Network remote	Trace stops, loops	traceroute -n
4 Transport	Timeout or refused	nc -vz, tcpdump
5/6 TLS	Cert or version error	openssl s_client
7 DNS	Name does not resolve	dig, dig @1.1.1.1
7 HTTP	403, 404, 500	curl -v -i
7 App auth	Permission refused	the service API

2. The same nine rows, each with a real example.

Layer	Real example of a failure
1 Physical	Wi-Fi at -88 dBm, rate collapses
2 Data link	arp -a prints (incomplete)
3 Network local	ping to 192.168.0.1 times out
3 Network remote	trace dies before hop 7
4 Transport	15 s of silence on port 443
5/6 TLS	Certificate expired yesterday
7 DNS	NXDOMAIN for a mistyped name
7 HTTP	403 on a push, token scope
7 App auth	Token lacks workflow scope

3. The last row is from the reader's own session. A push was refused because the personal access token did not carry the `workflow` scope and the push changed a CI workflow file.
4. That failure is as high in the stack as a failure can be. Every layer below it worked perfectly. The bytes arrived, were decrypted, were understood, and were then declined on policy grounds.

33.7.4 PLAIN - what is really happening inside

1. Two rules make the table usable. First, an error message names the layer that noticed, not the layer that failed. Git's sideband error was written at layer 7 about a layer 4 event.
2. Second, the higher the layer, the more specific the error text, and the more likely it is to be honest about itself.
3. A 403 from an API tells you precisely what is wrong. A timeout tells you almost nothing, because a timeout is what you get when nobody spoke.
4. So an ugly, detailed error message is good news. Silence is bad news, because the failure is low and nobody on the path will explain themselves.

33.7.5 TECHNICAL - the engineer's version

1. Exact commands, roughly one per layer, on macOS.

```
# 1 physical / 2 link
ifconfig en0 | grep -E 'status:|ether|inet '
system_profiler SPAirPortDataType | grep -A4 'Current Net'
# 2 link
arp -a -n | grep 192.168.0.1
sudo tcpdump -i en0 -n arp
# 3 network
netstat -rn -f inet | head
```

```
ping -c 3 -W 1000 192.168.0.1
traceroute -n -w 1 -q 3 20.207.73.82
# 4 transport
nc -vz -G 5 20.207.73.82 443
sudo tcpdump -i en0 -n 'host 20.207.73.82 and tcp'
# 5/6 TLS
openssl s_client -connect github.com:443 -servername github.com
# 7 application
dig +short github.com @1.1.1.1
curl -v -i --max-time 15 https://github.com
GIT_TRACE_PACKET=1 GIT_CURL_VERBOSE=1 git push
```

2. Interpretation notes that are easy to get wrong.
3. `ping` failing does not prove layer 3 is broken. Many networks and hosts drop ICMP echo by policy. Use it as positive evidence only.
4. `nc -vz` reporting "Connection refused" is a success for layers 1 to 4. It means a RST came back, so packets reach the host in both directions.
5. `openssl s_client` printing `verify error:num=20` means it could not build a chain to a trusted root, usually a missing intermediate certificate on the server, not an expired one. Read the number.
6. `curl` exit codes are layered too, which makes them a shortcut: 6 is DNS failure, 7 is connect failure, 28 is timeout, 35 is a TLS handshake error, 60 is certificate verification failure.

33.7.6 WORDS - remember these

1. Symptom — what you see — the observable behaviour, belonging to the layer that reported it, not necessarily the layer at fault.
2. NXDOMAIN — the name does not exist — the DNS response code for a queried name with no records at all.
3. Verify error 20 — the chain could not be built — OpenSSL's code for unable to get local issuer certificate, usually a missing intermediate.
4. Connection refused — a host said no — an RST in response to a SYN, which proves bidirectional reachability.
5. Scope — what a token is permitted to do — the permission set attached to a credential, checked at the application layer after everything else worked.

33.8 Where the model lies

33.8.1 PLAIN - in simple words

1. The layered model is a map. The internet is the territory. The map is wrong in several specific, well-known places.
2. Five places matter, and every working engineer meets all five.
3. First, TLS does not fit anywhere. It sits between transport and application and belongs to neither.
4. Second, a VPN carries whole layer 3 packets inside a layer 7 connection, so the layers stop being an order and become a loop.
5. Third, MPLS is so awkwardly placed that the industry gave up and invented the number 2.5 for it.
6. Fourth, NAT reaches into two layers at once and rewrites both, which the model says is forbidden.
7. Fifth, QUIC moved the transport layer out of the operating system and into ordinary programs.

33.8.2 PLAIN - a picture in your head

1. Think of a road map of a city printed in 1984.
2. But a tunnel has been dug that the map does not show, a one-way street has been reversed, and two roads have been merged into a flyover.

Where this comparison breaks:

3. A map can be reprinted. The OSI model cannot, because its value is that everybody already agreed on the numbers. Fixing it would destroy the only thing it is good for.

33.8.3 PLAIN – a worked example

1. The reader's own Mac showed several `utun` interfaces. Those are tunnel interfaces, and they are the clearest example of the model folding in on itself.
2. The VPN software takes that whole packet, encrypts it, and sends it as the payload of another connection, which has its own layer 3 and 4 headers.
3. So the order, reading outward, is: application, TLS, TCP, IP, and inside that, IP again, TCP again, TLS again, application again.
4. Layer 3 is now above layer 7. The model has no vocabulary for that.

33.8.4 PLAIN – what is really happening inside

1. The honest version: layers are not a physical fact about packets. They are a naming scheme for functions.
2. A packet does not know it is layer 3. It is a sequence of bytes that some piece of code chooses to read as an IP header.
3. Once you see that, tunnelling stops being strange. It is bytes carried by bytes.
4. And middleboxes stop being strange too. Nothing physically prevents a router from reading a TCP header, or a firewall from reading an HTTP path. The model says they should not. It cannot stop them.

33.8.5 TECHNICAL – the engineer's version

1. The five violations, precisely.
2. **TLS.** RFC 8446, August 2018, never assigns TLS an OSI layer number. TLS requires a reliable ordered byte stream below it and presents one above it, which makes it a transport-layer shim, but it performs no multiplexing and no addressing, which are the defining transport functions. Industry usage is inconsistent: load-balancer vendors say "layer 7", security vendors say "layer 6", researchers write "layer 4.5". None is normative.
3. **VPNs.** WireGuard, released by Jason Donenfeld in 2016 and merged into Linux 5.6 in March 2020, carries IP inside UDP inside IP: layer 3 over 4 over 3. OpenVPN, from James Yonan in 2001, can carry IP inside TLS inside TCP. IPsec, RFC 4301 from December 2005, carries IP inside IP. VXLAN, RFC 7348 from August 2014, carries Ethernet frames inside UDP: layer 2 over 4.
4. **MPLS.** RFC 3031, January 2001. The shim header is 4 bytes: a 20-bit label, 3 traffic class bits, 1 bottom-of-stack bit and an 8-bit TTL. It sits between the layer 2 and layer 3 headers, and forwarding uses the label instead of the IP address. It is neither layer 2 nor layer 3, so the industry calls it layer 2.5. That number appears in no standard.
5. **NAT.** RFC 3022, January 2001, updating RFC 1631 from May 1994. A NAT rewrites the layer 3 source address and the layer 4 source port together, then recomputes both the IP header checksum and the TCP or UDP checksum, because the transport checksum covers a pseudo-header containing the IP addresses. One device edits two layers in a single operation, and must understand the layer above it to do so at all.
6. The reader's own traffic was translated at least twice: once by the box at `192.168.0.1`, and again inside the provider's private core at the `172.31` and `172.26` addresses, which is carrier-grade NAT.
7. **QUIC.** RFC 9000, May 2021, with HTTP/3 in RFC 9114, June 2022. QUIC implements connection setup, reliability, ordering, flow control and congestion control, all transport functions, in user space over UDP.
8. Three consequences follow. Transport logic now ships and updates on the browser's schedule instead of the operating system's. Most of the transport header is encrypted, so middleboxes cannot read or modify it. And streams are independent, so one lost packet no longer stalls the others, which was TCP's head-of-line blocking problem.

9. Two more untidy cases. ARP has no clean layer: it uses its own EtherType rather than riding on IP, yet exists solely to serve IP. ICMP is carried inside IP yet is universally treated as part of layer 3.
10. Summary of the lies, with the honest placement.

Thing	Model says	Reality
TLS	Layer 5 or 6	Between 4 and 7
VPN tunnel	Strict ordering	Layer 3 inside layer 7
MPLS	No such layer	Called layer 2.5
NAT	One layer each	Edits 3 and 4 together
QUIC	Transport in kernel	Transport in the app

11. Keep the model anyway. Nobody has proposed a replacement that is both more accurate and as easy to talk about, and the numbers are how engineers communicate during incidents.

33.8.6 WORDS - remember these

1. Tunnel — carrying a whole packet inside another packet — encapsulating a layer N PDU inside a protocol at or above layer N.
2. Middlebox — a device in the path that reads above its own layer — NATs, firewalls, proxies and load balancers that inspect layer 4 or 7.
3. MPLS — forwarding by label instead of address — Multiprotocol Label Switching, RFC 3031, a 4-byte shim between layers 2 and 3.
4. TCP meltdown — two reliability timers fighting — throughput collapse from running TCP inside a TCP tunnel.
5. Head-of-line blocking — one lost piece stalling everything behind it — the TCP property that QUIC's independent streams remove.
6. Pseudo-header — why NAT must fix two checksums — the IP address fields included in the TCP and UDP checksum computation.

33.9 Why layering still earns its keep

33.9.1 PLAIN - in simple words

1. After a section listing what is wrong with the model, the fair question is whether it is worth keeping. It is, for three reasons. First, modularity: each piece can be built, tested and replaced on its own.
2. Second, independent innovation. Anyone can improve one layer without asking permission from the layers above or below.
3. Third, reasoning about faults. The whole of section 33.6 was only possible because the parts had names and an order.
4. The second reason is the one that built the internet. Nobody had to coordinate. A radio company could invent faster WiFi while a browser company invented a faster web, and neither had to speak to the other.

33.9.2 PLAIN - a picture in your head

1. Think of the standard shipping container, introduced in the 1950s.
2. The container is the agreed shape in the middle. Everything above and below it changed completely over seventy years. IP is that container.

Where this comparison breaks:

3. A container is a physical object with a fixed size. An IP packet is a convention that anyone can, and does, quietly violate with tunnels.

33.9.3 PLAIN - a worked example

1. Three real cases where one layer changed enormously and nothing above it needed to know.
2. Case one, layers 1 and 2. Ethernet ran at 10 Mbit/s in 1980 and now runs at 400 Gbit/s. That is 40,000 times faster.
3. The frame format did not change: six bytes of destination address, six of source, two of type. An IP stack written for 10 Mbit/s Ethernet describes the same frame that crosses a modern data centre.
4. Case two, layers 1 and 2 again, at home. The reader's household connection would have been a dial-up modem in the 1990s, then ADSL over a phone line, then fibre, and on a phone it is 4G or 5G radio.
5. Case three, the awkward TLS layer. TLS 1.3 was published as RFC 8446 in August 2018 and replaced almost everything about the handshake: two round trips became one, old ciphers were deleted, the negotiation was redesigned.
6. Web applications did not change. Servers and browsers were upgraded, and the HTTP requests flowing over the top were byte-for-byte identical.

33.9.4 PLAIN - what is really happening inside

1. The mechanism behind all three cases is a stable interface. As long as the service offered upward keeps its shape, the implementation beneath can be replaced entirely.
2. This is the same reason a program written in 1990 still runs on a modern processor: the instruction set is a stable interface, and everything under it was rebuilt many times.
3. It is also why badly drawn boundaries cause so much trouble. NAT broke the assumption that an address is stable end to end, and the internet is still paying for it thirty years later.

33.9.5 TECHNICAL - the engineer's version

1. Concrete figures for the three cases.

Change	Layer	Nothing above changed
10 Mbit/s to 400 Gbit/s	1 and 2	Same Ethernet frame
Dial-up to fibre to 5G	1 and 2	Same IP, same HTTP
TLS 1.2 to TLS 1.3	The shim	Same HTTP bytes

2. Ethernet's frame format is essentially unchanged since the DIX Ethernet version 2 specification of 1982 and IEEE 802.3, approved in 1983. Later additions such as the 4-byte 802.1Q VLAN tag from 1998 are insertions, not redesigns.
3. Wi-Fi shows the same pattern at layer 1: 802.11 in 1997 at 2 Mbit/s, 802.11b in 1999 at 11 Mbit/s, 802.11g in 2003 at 54 Mbit/s, 802.11n in 2009, 802.11ac in 2013, 802.11ax as Wi-Fi 6 in 2021, and Wi-Fi 7 certification from January

2024. The MAC frame the reader's laptop sends is recognizably the 1997 frame.

4. The counterexample proves the rule. IPv6, specified in 1998 and now RFC 8200 from July 2017, changed the layer everything else depends on, and adoption has taken three decades. The reader's machine reported `IPv6: (none)`.

33.9.6 WORDS - remember these

1. Modularity — building in replaceable pieces — decomposition into units with stable interfaces and hidden implementations.
2. Stable interface — the promise that does not change — the contract a layer offers upward, permitting its implementation to be replaced.
3. Narrow waist — the one thing everybody must agree on — the single universal convergence protocol, IP, in the internet architecture.
4. Ossification — the network becoming hard to change — the effect of middleboxes depending on details protocols never promised to keep.

33.10 The same idea outside networking

33.10.1 PLAIN - in simple words

1. Layering is not a networking idea. It is an engineering idea that networking happens to describe most carefully.
2. In graphics, a game does not talk to a graphics chip. It talks to a library, which talks to a driver, which talks to the chip.
3. In storage, a program does not talk to flash memory. It writes a file, which a filesystem turns into numbered blocks, which a driver turns into commands, which the drive turns into operations on cells.
4. In compilers, your source becomes tokens, then a tree, then a neutral intermediate form, then machine instructions, then something smaller still inside the processor.

33.10.2 PLAIN - a picture in your head

1. Think of a restaurant. You read a menu, not a recipe. The cook reads a recipe, not a farm. The farm grows food without knowing any menu.

Where this comparison breaks:

2. In a restaurant a customer can walk into the kitchen. In a well-built software stack, upper levels usually cannot reach past the level below, and when they can, that shortcut becomes a bug that lasts for years.

33.10.3 PLAIN - a worked example

1. The graphics stack, top to bottom, with the reader's Mac in mind.

```
Your program          draw this triangle
Graphics API          Metal, Vulkan, OpenGL calls
Driver                translates to GPU commands
Command buffer        a queue in shared memory
GPU hardware          thousands of small cores
```

2. The storage stack has the same shape.

```
Your program          write("notes.txt", data)
Filesystem            APFS or ext4: file -> blocks
Block layer           queue, scheduling, caching
Driver                NVMe or SATA commands
Flash translation     pretends to be numbered sectors
NAND cells            charge trapped in floating gates
```

3. The flash translation layer is the interesting one. It presents a lie upward: it pretends the drive is numbered sectors that can be overwritten in place. NAND flash cannot do that at all.
4. That is exactly what NAT does at layer 3, and it causes the same class of surprises: performance that does not match the model, and behaviour that only makes sense once you know about the hidden layer.

33.10.4 PLAIN - what is really happening inside

1. The compiler stack has the same shape. Source text becomes tokens, tokens become a tree, and the tree becomes an intermediate representation that belongs to no particular processor.
2. The value of the neutral middle form is the value of IP: many languages above it, many processors below it, one thing in the middle.

33.10.5 TECHNICAL - the engineer's version

1. Stable interfaces acting as narrow waists, outside networking.

Domain	The narrow waist	Since
Graphics	OpenGL, then Vulkan	1992, 2016
Compilers	LLVM IR	2003
Processors	The x86-64 instruction set	2000
Storage	The block device interface	1970s
Operating systems	The POSIX system calls	1988

2. LLVM began as Chris Lattner's work at the University of Illinois around 2000 to 2003, and its intermediate representation is now the middle layer for Swift, Rust, Clang and many others.
3. Direct3D appeared in 1995, Metal in 2014, Vulkan 1.0 in February 2016. All are the same architectural move: a stable command interface above a driver that changes with every hardware generation.
4. NVMe 1.0, published in March 2011, replaced the disk-shaped assumptions of older interfaces with a queue-based one while keeping the block abstraction above it unchanged. Filesystems did not need rewriting.
5. The instruction set architecture is the layer boundary that matters most in a processor: the same binary runs on chips from different companies with entirely different internal designs, which is precisely the layering promise.

33.10.6 WORDS - remember these

1. Abstraction layer — a level that hides the mess below — a module presenting a simplified interface over a more complex implementation.
2. Intermediate representation — the neutral middle form in a compiler — a machine-independent program form such as LLVM IR.
3. Flash translation layer — the part that makes flash look like a disk — the controller firmware mapping logical blocks to physical NAND pages.
4. Instruction set architecture — the contract between software and a chip — the documented instructions and behaviour a processor must implement.
5. Leaky abstraction — when the mess below shows through — an interface whose performance or failure behaviour reveals its implementation.

33.98 Common wrong ideas

1. Wrong: data physically travels down through the layers. Right: nothing moves. The message sits in one buffer with reserved space at the front, and each layer writes its header into that space and moves a pointer. On a Mac that buffer is an `mbuf`, on Linux an `sk_buff`.
2. Wrong: TCP/IP has seven layers. Right: the internet architecture stated in RFC 1122, October 1989, has four: link, internet, transport and application. Seven is OSI, a different model from a different organization.
3. Wrong: the OSI model is what the internet uses. Right: the internet uses protocols designed before OSI existed. IP was RFC 791 in September 1981; ISO 7498 appeared in 1984. OSI survives as vocabulary and as a teaching frame, not as the running system.
4. Wrong: layer 8 is a real layer, for the user. Right: there is no layer 8 in ISO 7498 or any standard. It is a joke that became a habit, and using it in a serious document marks you as guessing.
5. Wrong: TLS is layer 6. Right: no standard says so. RFC 8446 assigns no layer number. TLS sits between transport and application and is called 5, 6 or 4.5 by different vendors, all informally.
6. Wrong: the traceroute stopped, so that is where the fault is. Right: routers commonly refuse to send the ICMP time-exceeded replies traceroute needs. Stars at the end of a trace are normal. The reader's trace ended in stars after hop 12 and that fact alone proved nothing.

7. Wrong: a router forwards frames. Right: a router forwards packets. It discards the incoming layer 2 frame entirely and builds a new one for the next hop with different addresses. Only the layer 3 header survives end to end, minus the TTL.
8. Wrong: switches work at layer 3 because they have IP addresses. Right: the management address on a switch is for talking to the switch. Its forwarding decisions use MAC addresses, which is layer 2. A device that forwards on IP addresses is a router, whatever the box is labelled.
9. Wrong: an error message tells you which layer failed. Right: it tells you which layer noticed. Git's `send-pack: unexpected disconnect while reading sideband packet` is a layer 7 message describing a layer 4 event.
10. Wrong: if the model and the network disagree, the network is doing it wrong. Right: the model is a map. NAT, VPNs, MPLS and QUIC all break it, they are all deployed at enormous scale, and they are not going away.

33.99 Chapter summary in 20 lines

1. Layering splits network work into levels, where each level uses the level below without knowing how it works and serves the level above.
2. The benefit is that a whole level can be replaced and nothing above notices: radio to copper, dial-up to fibre, TLS 1.2 to TLS 1.3.
3. The cost is duplicated work, hidden information, header overhead, and faults that cross layers and belong to nobody. RFC 3439 from December 2002 says so in a section titled "Layering Considered Harmful".
4. OSI has seven layers: physical, data link, network, transport, session, presentation, application, handling bits, frames, packets, segments and data.
5. It came from ISO subcommittee 16 starting in 1977, was described by Hubert Zimmermann in 1980, and was published as ISO 7498 in 1984, also ITU-T X.200.
6. The OSI protocol suite lost to TCP/IP. GOSIP made it a US federal purchasing requirement in August 1990 and the mandate was relaxed in 1995.
7. Pieces of OSI did survive and you use them daily: X.509 certificates, ASN.1 encoding, IS-IS routing in carrier backbones, and LDAP descended from X.500.
8. Layers 5 and 6 barely map onto anything real. No mainstream internet protocol implements a separate session or presentation entity.
9. The TCP/IP model has four layers: link, internet, transport, application, stated normatively in RFC 1122, October 1989.
10. Its shape is an hourglass: many link technologies below, many applications above, and exactly one IP in the middle.
11. In the reader's session, layer 1 was the 5 GHz WiFi radio, layer 2 was the 802.11 frame and the ARP lookup for `192.168.0.1`, and layer 3 was every IP address from `192.168.0.1` to `20.207.73.82`, including the private ISP hops `172.31.0.17` and `172.26.x`.
12. Layer 4 was TCP to port 443, layers 5 and 6 were TLS, and layer 7 was DNS to `1.1.1.1`, HTTP, and git's own pack and sideband protocol.
13. Encapsulating one real 190-byte HTTP GET produced a 318-byte WiFi frame: 22 bytes of TLS, 32 of TCP, 20 of IP and 54 of 802.11, so 40.3 percent overhead and 59.7 percent payload.
14. At layer 1 the picture is worse. That frame occupies about 6 microseconds of a roughly 207-microsecond airtime slot, under 3 percent.
15. The reader's failure sits precisely at layer 4. Layers 1, 2 and 3 reached hop 12 inside Microsoft's network in Pune, layer 4 never completed its handshake, and layers 5 to 7 were never attempted.
16. The general rule: find the highest layer that demonstrably works and the lowest that demonstrably fails, and the fault is between them.
17. Each layer has a signature failure and a tool that tests it: `ifconfig`, `arp -a`, `ping`, `traceroute`, `nc -vz`, `openssl s_client`, `dig`, `curl`.

18. The model lies in five well-known places: TLS fits nowhere, VPNs put layer 3 inside layer 7, MPLS is called layer 2.5, NAT rewrites layers 3 and 4 together, and QUIC moved transport into user space over UDP.
19. Keep the model anyway. It is how engineers talk to each other during incidents, and no clearer replacement has been proposed.
20. The same pattern runs through graphics, storage and compilers, so learning it once pays for itself far outside networking.

34

CHAPTER

KB

Firewalls, VPNs, CDNs, IPv6 and the Gaps

34.0 What this chapter gives you

1. You will be able to say exactly what `localhost`, `127.0.0.1` and `0.0.0.0` each mean, and why `0.0.0.0` means two different things depending on where you write it.
2. You will be able to explain the three port ranges, why ports below 1024 need root on a UNIX machine, and why a busy machine can run out of ports even though it has 65,535 of them.
3. You will be able to describe what a firewall is, what it is not, and the difference between a stateless filter and a stateful one, with real rules written in `pf`, `iptables`, `nftables` and a cloud security group.
4. You will be able to explain the difference between DROP and REJECT well enough to diagnose a fault from the shape of the failure alone, and you will recognize the reader's own 15-second silence as the DROP signature.
5. You will be able to name the middleboxes sitting between you and any server, and say what each one can and cannot see.
6. You will be able to explain what a VPN really does at the packet level, choose between IPsec, OpenVPN and WireGuard for a reason, and state plainly what a commercial VPN does not protect you from.
7. You will be able to tell a forward proxy from a reverse proxy for the rest of your life, and write the SSH commands for local, remote and dynamic port forwarding without looking them up.
8. You will be able to describe how a CDN handles one request end to end, and explain anycast and DNS steering as two separate mechanisms that are constantly confused.
9. You will be able to compute the theoretical minimum round trip from India to the United States from the speed of light in glass, and say why buying more bandwidth will not fix a slow page.
10. You will be able to read and write IPv6 addresses, explain SLAAC and neighbour discovery, and say what `IPv6: (none)` in the reader's own output did and did not cost them.

34.1 localhost, 127.0.0.1 and 0.0.0.0

34.1.1 PLAIN – in simple words

1. Three things get confused constantly, and all three appear in error messages every day. We will separate them once and for all.
2. `127.0.0.1` is an address that means **this machine**. Data sent there never leaves the computer.
3. `localhost` is a **name** for that address. It is a word, not an address, and something has to turn it into an address first.
4. `0.0.0.0` is not really an address at all. It is a placeholder, and it means two different things in two different places.
5. When a program **listens** on `0.0.0.0`, it means "accept connections on every network the machine has".
6. When something **sends** to `0.0.0.0`, it means "this host, or an address I do not know yet".
7. That double meaning is the whole confusion. Same four numbers, opposite jobs, decided by whether you are listening or sending.

8. The practical result trips up almost every beginner. A server started on `127.0.0.1` works perfectly from the same machine and is completely invisible from any other machine.
9. That is not a bug and not a firewall. It is the address doing exactly what it says.

34.1.2 PLAIN - a picture in your head

1. Think of a large office building with a postal address on the street, and an internal mail system for moving envelopes between desks.
2. `127.0.0.1` is the internal mail system. You write a note, drop it in the internal tray, and it reaches another desk in the same building. It never goes near the street.
3. `localhost` is the phrase "internal mail". It is a word people use. The mail room has to look it up to know which tray that is.
4. Binding to `127.0.0.1` is like telling reception "I only accept internal mail". A courier from outside is turned away at the door.
5. Binding to `0.0.0.0` is like saying "I accept mail from any entrance the building has, including the loading bay and any door they add next year".

Where this comparison breaks:

6. In a real building, a courier could still walk in and hand you the envelope in person. On a computer there is no such route. A packet addressed to `127.0.0.1` that arrives from a cable is thrown away by the operating system before any program sees it.
7. And the office has one internal mail system. Your machine has sixteen million loopback addresses, all valid, all pointing at itself.

34.1.3 PLAIN - a worked example

1. Here is the loopback interface on a Mac, and the same idea on Linux.

```
$ ifconfig lo0
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
```

2. Notice the MTU is 16384 bytes, not the usual 1500. There is no cable to respect, so the operating system allows much bigger chunks.
3. Notice `::1`. That is the IPv6 version of `127.0.0.1`, and it matters more than people expect, as we will see in a moment.
4. Now start a web server two ways and look at the difference.

```
$ python3 -m http.server 8000 --bind 127.0.0.1
$ lsof -nP -iTCP:8000 -sTCP:LISTEN
COMMAND  PID  USER  FD  TYPE  NODE NAME
Python  4411   you   4u  IPv4  TCP  127.0.0.1:8000 (LISTEN)

$ python3 -m http.server 8000 --bind 0.0.0.0
$ lsof -nP -iTCP:8000 -sTCP:LISTEN
COMMAND  PID  USER  FD  TYPE  NODE NAME
Python  4498   you   4u  IPv4  TCP  *:8000 (LISTEN)
```

5. The * in the second output is the whole story. The star means "any local address". The first case names one address and only one.
6. From a phone on the same home WiFi, the first server is unreachable and the second answers on the laptop's address, for example `192.168.0.24:8000`.
7. Here is the modern trap. Ask for `localhost` and you may be handed `::1` before `127.0.0.1`.

```
$ curl -v http://localhost:8000/
*   Trying [::1]:8000...
* connect to ::1 port 8000 failed: Connection refused
*   Trying 127.0.0.1:8000...
* Connected to localhost (127.0.0.1) port 8000
```

8. If the server had been listening only on the IPv4 loopback and the client had given up after the first attempt, you would see "connection refused" from a server that is plainly running. This exact problem hit many Node.js users after version 17, released in October 2021, changed the default order in which resolved addresses are tried.

34.1.4 PLAIN - what is really happening inside

1. The loopback interface is a fake network card built into the operating system. It has no hardware, no driver for a chip, and no cable.
2. When a program sends to `127.0.0.1`, the kernel builds a normal IP packet, hands it to the loopback device, and the loopback device immediately hands it back up the receive path.
3. So the packet goes down the sending half of the stack and straight up the receiving half, inside the same machine, in microseconds.
4. Everything else still happens. There is a real TCP handshake, real sequence numbers, real ports. You can capture it with `tcpdump -i lo0`.
5. The whole block `127.0.0.0` to `127.255.255.255` is loopback, not just the one address. That is 16,777,216 addresses, and every one of them means "this machine".
6. On Linux you can use any of them straight away. Running one test service on `127.0.0.2:80` and another on `127.0.0.3:80` is a genuinely useful trick.
7. On macOS only `127.0.0.1` is configured by default, and you must add the others yourself with `sudo ifconfig lo0 alias 127.0.0.2 up`.
8. A packet from the outside world claiming to be from or to `127.x.x.x` is called a **martian packet**, and the kernel discards it. That rule is why binding to loopback is a real isolation boundary and not just a hint.
9. `0.0.0.0` as a listening address is a wildcard. The socket is not tied to any one address, so it accepts anything arriving on any interface, including interfaces created later, such as a VPN tunnel that comes up afterwards.
10. `/etc/hosts` is a plain text file that maps names to addresses, and it is consulted before DNS on almost every system. Two columns: the address first, then one or more names.

34.1.5 TECHNICAL - the engineer's version

1. `127.0.0.0/8` is reserved for loopback by RFC 1122, October 1989, section 3.2.1.3, which states that a host must never send a datagram with such an address outside itself, and must discard one that arrives from a link. RFC 6890, April 2013, records it in the special-purpose address registry.
2. `::1/128` is the IPv6 loopback, defined in RFC 4291. IPv6 spends exactly one address on it rather than a sixteen-million block.
3. `localhost` is a special-use domain name under RFC 6761, February 2013. Resolvers are supposed to answer it locally with a loopback address and never send it to a DNS server. Not every stack obeys this.
4. `0.0.0.0` is `INADDR_ANY` in the sockets API. Passing it to `bind()` gives a wildcard binding. In IPv6 the equivalent is `in6addr_any`, written `::`.
5. A dual-stack listener bound to `::` will, on Linux with `net.ipv6.bindv6only` set to 0, also accept IPv4 connections through IPv4-mapped addresses of the form `::ffff:192.168.0.24`. On some BSDs the default is the opposite. This is a per-system default, not a standard.
6. `0.0.0.0` as a destination or source means "this host on this network" per RFC 1122. A DHCP client sends its first DISCOVER from source `0.0.0.0` because it has no address yet. Most stacks quietly rewrite

a `connect()` to `0.0.0.0` as a connect to loopback, which is an implementation detail and not something to rely on.

7. `0.0.0.0/0` in a routing table is the default route: the least specific match possible, used when nothing else matches.

Where written	Meaning
bind(0.0.0.0)	all local addresses
source 0.0.0.0	I have no address yet
route 0.0.0.0/0	default route
connect(0.0.0.0)	usually loopback

8. The `/etc/hosts` file is the direct descendant of `HOSTS.TXT`, the single file distributed by the Stanford Research Institute Network Information Center before DNS existed. On Linux the order of name sources is set in `/etc/nsswitch.conf`, usually `hosts: files dns`. On Windows the file is at `C:\Windows\System32\drivers\etc\hosts`. On macOS `mDNSResponder` reads `/etc/hosts` and it takes precedence over DNS.
9. Docker is where this bites hardest. Inside a container, `127.0.0.1` is the container's own loopback namespace, not the host's. A server bound to `127.0.0.1` inside a container is unreachable even with `-p 8080:8080`, because the published port forwards to the container's external address, not its loopback. Bind to `0.0.0.0` inside containers.
10. The reverse form `-p 127.0.0.1:8080:8080` publishes the port only on the host's loopback, which is the correct way to expose a database container to the host and to nothing else.
11. Observation commands: `lsof -nP -iTCP -sTCP:LISTEN` on macOS, `ss -tlnp` on Linux, `netstat -ano | findstr LISTENING` on Windows, `dscacheutil -q host -a name localhost` to see what macOS resolves.

34.1.6 WORDS - remember these

1. Loopback — a fake network card inside the machine — a virtual interface that returns transmitted packets to the local receive path.
2. localhost — the word for "this machine" — a special-use domain name reserved by RFC 6761, resolving to `127.0.0.1` or `::1`.
3. Wildcard bind — listen on everything — binding a socket to `INADDR_ANY`, `0.0.0.0`, or `in6addr_any`, `::`.
4. Martian packet — an impossible address arriving from outside — a datagram whose source or destination is reserved for local use, discarded on ingress.
5. Hosts file — a small local address book — a text file of address-to-name mappings consulted ahead of DNS.

34.2 Ports, properly this time

34.2.1 PLAIN - in simple words

1. An address gets a packet to a machine. A **port** gets it to the right program on that machine.
2. A port is just a number between 0 and 65,535, carried in the TCP or UDP header. It is not a physical thing.
3. The numbers are split into three groups by agreement, and the groups have different rules.
4. The first group, 0 to 1023, holds the famous ones: 22 for SSH, 80 for plain web, 443 for secure web, 53 for DNS.
5. On Mac and Linux, only an administrator may open a program on those low numbers. That rule is old and it exists for a reason we will explain.
6. The second group, 1024 to 49151, is for registered services. Anyone may use these. Databases and development servers live here.

7. The third group, 49152 upward, is for temporary use. Every time your machine makes an outgoing connection, it borrows one number from this pool, uses it, and gives it back.
8. That borrowed number is why two browser tabs can both talk to the same website without the replies getting mixed up. Each tab's connection has a different borrowed number.

34.2.2 PLAIN – a picture in your head

1. Think of a company with one street address and a switchboard with numbered extensions.
2. The street address is the IP address. The extension is the port. The switchboard is the operating system.
3. Extensions 1 to 1023 are reserved for official departments: reception, accounts, security. Only management may assign them.
4. Extensions in the middle range belong to named teams who have registered their number in the company directory.
5. The top range is a pool of temporary extensions. When someone in the building phones out, the switchboard lends them a spare extension so the reply comes back to the right desk.
6. When the call ends, the extension goes back in the pool, but only after a short cooling-off period, so a late reply to the old call does not ring a new person's phone.

Where this comparison breaks:

7. A real switchboard could reuse the same temporary extension for calls to two different companies at once. A computer can too, and that is the part people get wrong. The uniqueness rule is not about the port alone. It is about the whole set of four values: your address, your port, their address, their port.
8. Also, a phone extension has one line. A listening port can hold thousands of simultaneous connections, all with the same local port number.

34.2.3 PLAIN – a worked example

1. Take the reader's own failed request. macOS lends an ephemeral port from its range, and the connection is described by four values.

```
Source address: 192.168.0.24     (the laptop on the home LAN)
Source port:    54318            (borrowed from 49152-65535)
Dest address:   20.207.73.82     (github.com, as resolved)
Dest port:      443              (HTTPS)
```

2. That set of four is the **four-tuple**. It is what makes the connection unique on the machine. The kernel matches arriving packets against it.
3. Now the exhaustion arithmetic, which is the thing worth remembering. Suppose a Linux server opens many short-lived connections to one database at one address and one port.
4. Linux offers 32768 to 60999 by default, which is 28,232 ports. All four other values in the tuple are fixed, so the port number is the only thing that can vary.
5. After each connection closes, the port sits in `TIME_WAIT` for 60 seconds on Linux before it can be reused.
6. So the sustainable rate is 28,232 divided by 60, which is about 470 new connections per second to that single destination. Above that, `connect()` starts failing with `EADDRNOTAVAIL`.
7. The fix is not a bigger port range. The fix is connection reuse: a pool that keeps connections open instead of opening a new one per query.
8. Here is what listening sockets look like on a real machine.

```
$ ss -tlnp
State  Local Address:Port   Peer     Process
LISTEN 127.0.0.1:5432       0.0.0.0:* postgres
LISTEN 0.0.0.0:8080         0.0.0.0:* java
LISTEN [::]:443             [::]:*    nginx

$ netstat -an -p tcp | head -4
```

```
Proto  Local Address       Foreign Address      (state)
tcp4   192.168.0.24.54318  20.207.73.82.443     SYN_SENT
tcp4   127.0.0.1.5432      *.*                  LISTEN
```

9. That `SYN_SENT` line is the reader's fault frozen in place. The connection was attempted, the port was allocated, and no reply ever changed the state.

34.2.4 PLAIN - what is really happening inside

1. When you call `connect()` without choosing a port, the kernel picks one for you. This is called an **ephemeral** or dynamic port.
2. Old systems picked the next number in sequence. That was predictable, so modern systems randomize the choice to make blind attacks harder.
3. The kernel then checks that the resulting four-tuple is not already in use. If it is, it picks again.
4. When a connection closes, the side that closed first keeps the tuple in a waiting state called `TIME_WAIT`.
5. The reason is simple and important. Packets from the old connection may still be wandering the network. If the same tuple were reused immediately, a straggler could be accepted as part of the new connection.
6. The second reason is that the last acknowledgement might be lost, and the other side may resend its final packet. Something must be there to answer it.
7. The privileged port rule is a convention from early UNIX. On a machine where only an administrator could bind port 513, receiving a connection from a low port was weak evidence that a trusted program on a trusted machine sent it.
8. That trust model is long dead, because anyone can run their own machine and bind whatever they like. The restriction survives anyway, and now it mainly stops an ordinary user from hijacking the web server's port.
9. `SO_REUSEADDR` is the socket option that says "let me bind this port even though an old connection is still cooling off". Without it, restarting a server often fails with "address already in use" for up to a minute.

34.2.5 TECHNICAL - the engineer's version

1. Ports are 16-bit unsigned fields in the TCP and UDP headers, so 0 to 65,535. Port 0 is reserved; binding to it asks the kernel to choose.
2. RFC 6335, August 2011, also known as BCP 165, defines the registry and the three ranges, jointly maintained by IANA and the IETF.

Range	Name	Rule
0-1023	System, well-known	IETF review to assign
1024-49151	User, registered	IANA assignment
49152-65535	Dynamic, private	never assigned

3. The privileged range on UNIX is enforced in the kernel. On Linux the check is against `CAP_NET_BIND_SERVICE`, and since kernel 4.11 in 2017 the threshold itself is tunable with `net.ipv4.ip_unprivileged_port_start`, default 1024. Windows has never had this restriction at all.
4. The actual ephemeral ranges in use do not match the IANA suggestion, and this surprises people. Real defaults:

System	Ephemeral range	Count
Linux	32768-60999	28,232
macOS, recent	49152-65535	16,384

System	Ephemeral range	Count
Windows Vista+	49152-65535	16,384
FreeBSD	10000-65535	55,536

5. Check them with `sysctl net.ipv4.ip_local_port_range` on Linux, `sysctl net.inet.ip.portrange.first net.inet.ip.portrange.last` on macOS, and `netsh int ipv4 show dynamicport tcp` on Windows.
6. RFC 6056, January 2011, specifies port randomization algorithms, motivated by off-path attacks against TCP and against DNS after Dan Kaminsky's 2008 disclosure.
7. `TIME_WAIT` is defined in RFC 793, September 1981, as twice the maximum segment lifetime, with an MSL of 2 minutes, giving 4 minutes. Linux ignores that and hardcodes 60 seconds in `TCP_TIMEWAIT_LEN`, a compile-time constant that no sysctl exposes.
8. `SO_REUSEADDR` permits binding a local address that is in `TIME_WAIT`. `SO_REUSEPORT`, added in Linux 3.9 in April 2013, is different: it lets several sockets bind the identical address and port so the kernel can spread incoming connections across worker processes. On BSD the two options have older and slightly different semantics.
9. `net.ipv4.tcp_tw_reuse` allows reuse of a `TIME_WAIT` entry for a new outgoing connection when timestamps confirm safety. Its dangerous cousin `tcp_tw_recycle` broke clients behind NAT and was removed from Linux in version 4.12, July 2017. If you find advice recommending it, the advice is out of date.
10. Diagnostic commands worth memorizing:

```
lsof -nP -iTCP -sTCP:LISTEN        # macOS: who is listening
lsof -nP -i :8080                  # who owns port 8080
ss -tan state time-wait | wc -l    # Linux: TIME_WAIT count
netstat -an -p tcp | grep 443      # macOS: all port 443 sockets
sysctl net.inet.ip.portrange.first # macOS ephemeral start
```

34.2.6 WORDS - remember these

1. Port — a number picking the program — a 16-bit field in the TCP or UDP header identifying an endpoint within a host.
2. Ephemeral port — a temporary number borrowed for one outgoing call — a dynamically allocated source port from the local dynamic range.
3. Four-tuple — the four values that make a connection unique — source address, source port, destination address, destination port.
4. TIME_WAIT — the cooling-off period after a connection ends — a TCP state held by the closing side for twice the maximum segment lifetime.
5. Privileged port — a low number only an administrator may open — ports below 1024, requiring root or `CAP_NET_BIND_SERVICE` on UNIX systems.
6. Port exhaustion — running out of temporary numbers — failure of `connect()` with `EADDRNOTAVAIL` when no free four-tuple exists.

34.3 Firewalls, properly

34.3.1 PLAIN - in simple words

1. A firewall is a set of rules that decides which packets may pass and which are stopped. That is the entire idea.
2. It sits somewhere on the path: in your laptop, in your home router, at the edge of a company, or in front of a cloud server.
3. What a firewall is not: it is not antivirus, it does not read your files, and it does not know whether a program is good or evil.

4. It also cannot protect you from something you invited in. If you connect out to a hostile server, the firewall was not asked.
5. The simplest kind looks at each packet alone and asks: where is it from, where is it going, which port, which protocol.
6. A smarter kind remembers connections. When you start a conversation, it notes that fact, and it lets the replies back in without a separate rule.
7. That memory is called **state**, and it is the single biggest idea in firewalls.
8. Two directions matter. **Ingress** filtering controls what comes in. **Egress** filtering controls what goes out. Most people only think about the first.
9. Egress filtering is what stops a compromised machine from phoning home. It is also, occasionally, what stops you reaching a site for no reason you can see.

34.3.2 PLAIN - a picture in your head

1. Picture a security desk at the entrance to a building.
2. The stateless guard has a printed list: "let in anyone wearing a blue badge, turn away everyone else". Each person is judged alone, with no memory.
3. If you send a courier out with a parcel and the courier returns with a reply, the stateless guard has no idea. He checks the printed list again.
4. So with a stateless guard you need a second rule saying "let returning couriers in", and that rule is a hole, because you cannot tell a real returning courier from someone pretending.
5. The stateful guard keeps a notebook. When your courier leaves, he writes down who left, where they went and when.
6. When a reply arrives that matches a line in the notebook, it is let in automatically. Nothing else is.
7. He crosses out lines after a while, so the notebook does not grow forever.

Where this comparison breaks:

8. The guard's notebook holds a hundred lines. A firewall's table holds hundreds of thousands, and when it fills, new connections are silently dropped. That failure looks exactly like a network outage and is one of the nastiest faults in operations.
9. And the guard can see faces. A firewall in front of an HTTPS site cannot see the contents at all, only the outside of the envelope.

34.3.3 PLAIN - a worked example

1. The same policy, "allow SSH and HTTPS in, allow everything out, drop the rest", written four ways on real systems.

```
# Linux, iptables
iptables -P INPUT DROP
iptables -A INPUT -i lo -j ACCEPT
iptables -A INPUT -m conntrack \
         --ctstate ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -p tcp --dport 22  -j ACCEPT
iptables -A INPUT -p tcp --dport 443 -j ACCEPT
```

```
# Linux, nftables
nft add table inet filter
nft add chain inet filter input \
    '{ type filter hook input priority 0; policy drop; }'
nft add rule inet filter input ct state established,related accept
nft add rule inet filter input iif lo accept
nft add rule inet filter input tcp dport { 22, 443 } accept
```

```
# macOS and BSD, /etc/pf.conf
set skip on lo0
```

```
block drop in all
pass out all keep state
pass in proto tcp to any port { 22, 443 } keep state
```

```
# AWS security group, one ingress rule
aws ec2 authorize-security-group-ingress \
  --group-id sg-0abc123 --protocol tcp \
  --port 443 --cidr 0.0.0.0/0
```

2. Read the common shape. Every one of them: default deny inbound, allow loopback, allow replies to things we started, then a short list of open ports.
3. In the cloud version there is no "allow established" line, because AWS security groups are stateful by design and you cannot make them otherwise.
4. Look at the connection tracking table itself on a Linux box.

```
$ sudo conntrack -L | head -3
tcp 6 431999 ESTABLISHED src=10.0.1.5 dst=20.207.73.82
    sport=54318 dport=443 [ASSURED] mark=0 use=1
tcp 6 118 TIME_WAIT src=10.0.1.5 dst=10.0.2.9
    sport=41022 dport=5432 [ASSURED] mark=0 use=1
```

5. The number after the protocol is the seconds left on that entry. 431,999 seconds is just under five days, which is the default timeout for an established TCP flow on Linux, and it is far too long for a busy box.

34.3.4 PLAIN - what is really happening inside

1. A stateless filter is a loop. For each packet, walk the rule list from the top, compare fields, and act on the first rule that matches.
2. The fields compared are the same handful every time: protocol number, source address, destination address, source port, destination port, and TCP flags.
3. A stateful filter does that only for the first packet of a flow. If the packet is allowed, it writes an entry in a table.
4. The entry holds the four-tuple, the protocol, the current TCP state, byte counters, and a timer.
5. Every later packet is looked up in that table first, usually with a hash, in constant time. Only unmatched packets fall through to the rules.
6. That is why stateful firewalls are fast for long connections and expensive for floods of new ones. A SYN flood attacks the table, not the rules.
7. For UDP, which has no connections, the firewall invents one. It creates an entry when it sees an outgoing datagram and expires it after a short timeout, usually 30 seconds. This is why your video call needs regular traffic to keep working.
8. An application-layer firewall goes further and parses the payload. It knows what an HTTP request looks like, so it can act on a method, a path or a header.
9. It cannot do that if the traffic is encrypted, unless it also terminates the encryption, which means installing its own certificate authority on every client. That is a large decision, not a checkbox.

34.3.5 TECHNICAL - the engineer's version

1. The vocabulary, ordered by how much the device understands:

Kind	Sees	Typical example
Stateless filter	one packet	router ACL, AWS NACL
Stateful filter	flows	pf, iptables, sec group
Proxy firewall	full protocol	TIS FWTK, Squid
Next generation	app and user	Palo Alto, Fortinet
Web app firewall	HTTP semantics	ModSecurity, AWS WAF

Kind	Sees	Typical example

2. History, with names and dates. Jeff Mogul's `screend` at Digital Equipment Corporation in 1989 was an early packet filter. DEC SEAL in 1991, built by Marcus Ranum, is widely called the first commercial firewall. Ranum's TIS Firewall Toolkit followed in 1993.
3. Stateful inspection was commercialized by Check Point in FireWall-1, 1994, and patented by Gil Shwed, United States patent 5,606,668, filed December 1993 and granted February 1997.
4. William Cheswick and Steven Bellovin published "Firewalls and Internet Security: Repelling the Wily Hacker" in 1994, which is where much of the modern vocabulary comes from.
5. Palo Alto Networks was founded in 2005 by Nir Zuk, and the term next-generation firewall was popularized by Gartner around 2009. It means stateful inspection plus application identification, user identity and intrusion prevention in one device. It is a marketing category, not a standard.
6. ModSecurity, the first widely used open web application firewall, was released by Ivan Ristic in 2002. The OWASP Core Rule Set is the common rule pack.
7. Linux lineage: `ipfwadm` in kernel 2.0, 1996; `ipchains` in 2.2, January 1999; `iptables` with the netfilter framework by Rusty Russell in 2.4, January 2001; `nftables` merged in kernel 3.13, January 2014. Debian 10 in 2019 made `nftables` the default backend.
8. BSD lineage: `pf` was written by Daniel Hartmeier for OpenBSD 3.0 in December 2001, after a licence dispute over IPFilter. Apple adopted `pf` in Mac OS X 10.7 Lion, July 2011. macOS also has a separate per-application firewall, controlled by `socketfilterfw`, present since Mac OS X 10.5 Leopard in 2007.
9. Windows: Internet Connection Firewall shipped in Windows XP in 2001, was enabled by default and renamed Windows Firewall in XP Service Pack 2, August 2004, gained outbound rules in Vista in 2007, and was renamed Microsoft Defender Firewall around 2019.
10. Connection tracking limits are real operational numbers. `nf_conntrack_max` is sized from RAM at boot and is often between 65,536 and 262,144. `nf_conntrack_tcp_timeout_established` defaults to 432,000 seconds. When the table fills, the kernel logs "nf_conntrack: table full, dropping packet" and new connections vanish without explanation.
11. Cloud: an AWS security group is stateful, allows only `allow` rules, and is evaluated as the union of all groups attached to an interface. A network ACL is stateless, has numbered `allow` and `deny` rules evaluated in order, and requires you to open the ephemeral range 1024-65535 for return traffic. Both fail silently, which matters enormously in section 34.4.
12. Observation: `sudo pfctl -sr` and `pfctl -si` on macOS, `iptables -L -n -v` and `nft list ruleset` on Linux, `netsh advfirewall firewall show rule name=all` on Windows, `conntrack -L` and `cat /proc/sys/net/netfilter/nf_conntrack_count`.

34.3.6 WORDS - remember these

1. Firewall — a rule-based gate for packets — a policy enforcement point filtering traffic on header fields, flow state or application content.
2. Stateless filter — judges each packet alone — a filter with no flow table, requiring explicit rules for return traffic.
3. Stateful inspection — remembers conversations — filtering using a connection tracking table keyed on the flow tuple and protocol state.
4. Connection tracking table — the firewall's notebook — the in-memory table of active flows, with states, timers and counters.
5. Ingress and egress — inbound and outbound filtering — policy applied to traffic entering or leaving an interface or network boundary.
6. Web application firewall — a filter that reads HTTP — a layer 7 filter matching request patterns against attack signatures.

34.4 DROP versus REJECT

34.4.1 PLAIN - in simple words

1. When a firewall decides not to let a packet through, it has two ways to behave, and the difference decides how your day goes.
2. **REJECT** means: refuse, and say so. The firewall sends a short message back saying "no".
3. **DROP** means: refuse, and say nothing. The packet is deleted. No reply of any kind is generated.
4. To the client, REJECT is a closed door with a sign on it. DROP is a wall where you expected a door.
5. With REJECT, your program fails almost instantly, usually in well under a second, with a clear error like "connection refused".
6. With DROP, your program has no idea anything happened. It assumes the request got lost, waits, sends it again, waits longer, sends it again.
7. So it hangs. Fifteen seconds, thirty seconds, seventy-five seconds, depending on the system and the program, and then it gives up with "timed out".
8. Both outcomes are a refusal. Only one of them tells you so.
9. This is the exact shape of the reader's own fault, and we will come back to it at the end of this section.

34.4.2 PLAIN - a picture in your head

1. You knock on a door in a large building looking for an office.
2. In the REJECT case, someone opens the door and says "wrong floor". You know immediately. Total time: two seconds. You go and look elsewhere.
3. In the DROP case, nobody answers. You wait. You knock again, louder. You wait longer. You knock a third time.
4. Eventually you leave. And here is the key point: you still do not know whether the office is empty, whether the person is ignoring you, or whether you were at the wrong building all along.
5. That is the whole cost of silence. The refusal carries no information.
6. Now think about why a building might do this on purpose. Someone walking the corridor knocking on every door learns nothing from silence. If every wrong door said "wrong floor", they would map the building in a minute.

Where this comparison breaks:

7. A person knocking gives up in a minute. A computer keeps a strict schedule of retries with doubling gaps, and the total is predictable to the second.
8. And in a building the silence comes from one identifiable door. On a network, the silence could come from any of a dozen devices on the path, and you cannot tell which, because none of them left a trace.

34.4.3 PLAIN - a worked example

1. The same request against three different policies, with real timings.

```
Case 1: REJECT with a TCP reset
$ time curl -v https://example.test
*   Trying 203.0.113.9:443...
* connect to 203.0.113.9 port 443 failed: Connection refused
real  0m0.031s

Case 2: REJECT with ICMP administratively prohibited
$ time curl -v https://example.test
*   Trying 203.0.113.9:443...
* connect to 203.0.113.9 port 443 failed: No route to host
real  0m0.042s

Case 3: DROP, nothing sent back
$ time curl -v https://example.test
```

```
*   Trying 203.0.113.9:443...
* Connection timed out after 15001 milliseconds
real  0m15.004s
```

2. Case 3 is the reader's session, character for character. `curl -v` printed `Trying 20.207.73.82:443...` and then produced nothing for 15 seconds.
3. Now the packets on the wire in case 3. The client alone is talking.

```
 0.000 192.168.0.24.54318 > 20.207.73.82.443: Flags [S]
 1.001 192.168.0.24.54318 > 20.207.73.82.443: Flags [S]
 3.005 192.168.0.24.54318 > 20.207.73.82.443: Flags [S]
 7.010 192.168.0.24.54318 > 20.207.73.82.443: Flags [S]
15.020 192.168.0.24.54318 > 20.207.73.82.443: Flags [S]
(no packets in the other direction, ever)
```

4. The gaps double: one second, two, four, eight. That doubling is the standard retransmission backoff. Nothing in the other direction is the signature.
5. Compare with the REJECT capture, which is two packets and over.

```
 0.000 192.168.0.24.54318 > 203.0.113.9.443: Flags [S]
 0.028 203.0.113.9.443 > 192.168.0.24.54318: Flags [R.]
```

34.4.4 PLAIN - what is really happening inside

1. A REJECT is an active act. The firewall generates a brand-new packet and sends it back to the source address.
2. For TCP it usually sends a reset, which is a TCP packet with the RST flag set. The client's kernel sees it and immediately fails the `connect()` call with "connection refused".
3. Alternatively it sends an ICMP error, a small control message. The most honest one is "communication administratively prohibited", which literally means "a policy stopped this".
4. The client's kernel maps that ICMP error to an error code and the program reports something like "no route to host" or "host unreachable".
5. A DROP does nothing at all. The packet is freed. No counter is incremented in anything the client can see. No log line reaches the client.
6. So the client's TCP stack sits in a state called `SYN_SENT` and follows its retransmission schedule, doubling the wait each time.
7. There is a second, subtler reason silence is not always deliberate. A device under heavy load may rate-limit the errors it generates, so a genuine REJECT can be swallowed and look like a DROP.
8. And a black hole, where a router has no route and the error is filtered on the way back, produces the same silence with nobody having intended it.
9. That is why the honest reading of silence is: something on this path is not delivering, and it did not say why. Not: someone blocked me.

34.4.5 TECHNICAL - the engineer's version

1. The client-visible symptom table. Learn this and you can diagnose from the error message alone.

Cause	Client sees	Typical time
Port closed, host up	ECONNREFUSED	under 50 ms
REJECT with tcp-reset	ECONNREFUSED	under 50 ms
REJECT with ICMP 3/13	EHOSTUNREACH	under 100 ms
DROP, silent	ETIMEDOUT	15 to 130 s
No route locally	ENETUNREACH	instant
Wrong host, alive	TLS or 404 error	one RTT

2. The ICMP codes that carry a refusal are all type 3, destination unreachable: code 0 network unreachable, code 1 host unreachable, code 3 port unreachable, code 9 network administratively prohibited, code 10 host administratively prohibited, code 13 communication administratively prohibited. In ICMPv6 the equivalent is type 1 code 1.
3. `iptables -j REJECT` defaults to `--reject-with icmp-port-unreachable`, which is type 3 code 3. `--reject-with tcp-reset` sends an RST instead, and is the politest option for TCP because it is indistinguishable from a closed port. `--reject-with icmp-admin-prohibited` is the most informative and the least used.
4. In `pf` the two spellings are `block drop` and `block return`, and `block return-rst` or `block return-icmp` name the reply explicitly.
5. `nmap` names the three outcomes directly, and this is the cleanest vocabulary there is: `open` means a SYN-ACK came back, `closed` means an RST came back, `filtered` means nothing came back. A `filtered` result is a DROP by definition.
6. Timeouts differ by system, and the reader's 15 seconds came from `curl`, not from the kernel:

Layer	Setting	Default
Linux kernel	tcp_syn_retries	6, about 127 s
macOS kernel	net.inet.tcp.keepinit	75000 ms
curl	–connect-timeout	none by default
Go net.Dialer	Timeout	none unless set

7. Why operators choose DROP, stated fairly. It denies a scanner confirmation that a host exists. It forces the scanner to wait for a timeout on every probe, turning a one-second sweep into a multi-hour job. It generates no traffic, so it cannot be used to reflect or amplify an attack at a third party. And it does not reveal that a filtering policy exists at all.
8. Why DROP is expensive for everyone else. Every failure mode collapses into one symptom. A dropped SYN, a routing black hole, an asymmetric return path, a full conntrack table, a dead server behind a load balancer and a national filter all produce the identical client experience.
9. Cloud platforms make this worse by having no choice. AWS security groups and network ACLs both drop silently. There is no reject option. So a one-line security group mistake presents to the developer as an unexplained 15-second hang, which they will spend an hour blaming on DNS.
10. Now the tie back, stated precisely, separating proof from suggestion.

Observation	What it proves
DNS returned 20.207.73.82	resolution worked
SYNs sent, nothing back	silent drop on the path
Same request fine on mobile	not the server, not DNS
Traceroute ends in stars	nothing on its own

11. What is proven: packets left the machine towards `20.207.73.82:443` and nothing at all came back for 15 seconds, and the same request over a different access network succeeded. That is the DROP signature and it rules out the destination being down.
12. What is not proven: which device dropped, whether it was on the forward or the return path, whether it was policy or fault, and whether it was permanent. A silent drop leaves no evidence of its author, which is exactly what its author intended.

34.4.6 WORDS - remember these

1. DROP — discard and say nothing — a filtering action that frees the packet without generating any response.
2. REJECT — refuse and say so — a filtering action that generates a TCP RST or an ICMP destination-unreachable message.
3. Administratively prohibited — a policy said no — ICMP type 3 code 13, or ICMPv6 type 1 code 1.
4. Filtered — the scanner's word for silence — an `nmap` port state meaning no response was received to any probe.
5. Black hole — traffic that vanishes with no error — a path that discards packets and returns no ICMP, whether by policy or by fault.

34.5 Middleboxes, and the end of end-to-end

34.5.1 PLAIN - in simple words

1. The simple story of the internet says two computers talk directly, and the routers in between only forward.
2. That story is no longer true anywhere. Between you and any server sits a row of machines that change, inspect or block what passes.
3. These machines have a collective name: **middleboxes**.
4. Some rewrite addresses so many homes can share one public address. That is NAT, and the reader's own trace showed private ISP addresses at hops 2 to 6.
5. Some quietly capture your web requests and answer from a local copy, without you configuring anything. Those are transparent proxies.
6. Some look inside packets to identify what application you are using, and act on it. That is deep packet inspection.
7. Some break your encryption on purpose, read the contents, and re-encrypt with their own certificate. That is TLS interception, common inside companies.
8. Some hijack your very first web request and show you a login page. That is a captive portal, the thing at every airport and hotel.
9. The pattern is always the same: something in the middle knows more about your traffic than the simple model allows.

34.5.2 PLAIN - a picture in your head

1. You post a letter to a friend and imagine it travelling sealed from your hand to theirs.
2. In reality the sorting office takes your envelope, puts it in a new envelope with the office's own return address, and forwards it. That is NAT.
3. Another office keeps photocopies of popular letters and, if someone asks for one, sends the copy instead of forwarding the request. That is a transparent cache.
4. A third reads the address, the size, the thickness and the postmark to guess what the letter contains, without opening it. That is deep packet inspection on encrypted traffic.
5. A fourth actually opens the envelope, reads it, copies it into a fresh envelope, and reseals it with its own wax stamp. Your friend sees a sealed letter and may not notice the stamp changed. That is TLS interception.
6. A fifth holds every letter until you sign a form at the counter. That is a captive portal.

Where this comparison breaks:

7. In the post, opening a sealed letter is illegal and visible. On a network, TLS interception is legal, routine on company laptops, and technically invisible unless you look at the certificate chain.
8. And a letter passes maybe three offices. A packet from an Indian flat to a server in the United States passes twenty or more devices, several of which have inspection features switched on.

34.5.3 PLAIN – a worked example

1. Here is the real path from the reader's session, annotated with which kind of box each hop is likely to be. Everything marked "likely" is inference, not proof.

```
hop  address              likely role
 1   192.168.0.1          home router: NAT, DHCP, DNS, firewall
 2   172.31.0.17          ISP access, private address
 3   137.97.29.249        ISP public edge
 4   172.26.22.235        ISP core, private address
 5-6 172.16.18.33 etc     ISP core, load balanced links
 7   104.44.196.187       Microsoft edge, Delhi
 8-9 104.44.55.163 etc    Microsoft, Mumbai
10-12 104.44.31.62 etc    Microsoft, Pune
13+  * * *                no replies
```

2. Two NATs are visible in that list, not one. The home router translates `192.168.0.24` to whatever address it holds, and the presence of RFC 1918 private addresses in the ISP core is consistent with carrier-grade NAT translating again.
3. So the source address that Microsoft's edge saw was not the reader's home router address either. Two rewrites, neither of them visible from the laptop.
4. To detect TLS interception on any machine, look at who signed the certificate. A public site should chain to a public certificate authority.

```
$ openssl s_client -connect example.com:443 </dev/null 2>/dev/null \
  | openssl x509 -noout -issuer
issuer=C=US, O=DigiCert Inc, CN=DigiCert TLS RSA SHA256 2020 CA1

# On an intercepted network you would instead see something like:
issuer=C=IN, O=ExampleCorp IT, CN=ExampleCorp Root CA
```

5. If the issuer is your employer, every HTTPS page you load on that machine is readable by your employer's proxy. This is not a hack. It is a purchased product, and the certificate was installed by the device management tool.

34.5.4 PLAIN – what is really happening inside

1. A transparent proxy is put in the path by routing, not by configuration. The network is arranged so packets to port 80 or 443 arrive at the proxy instead of the real destination.
2. The proxy then opens its own connection onward and copies bytes between the two. Your operating system never knew.
3. Deep packet inspection on unencrypted traffic simply reads the payload. On encrypted traffic it cannot, so it reads what is still visible: the destination address, the port, the packet sizes and timing, and above all the server name sent in the clear at the start of a TLS handshake.
4. That server name field is the single most used censorship and filtering hook in the world today. It is plain text, it names the site, and it arrives before any encryption is in force.
5. TLS interception works by installing an extra root certificate on your machine. The proxy then generates a fresh certificate for every site you visit, signed by that root, and your browser trusts it because the root is in the trust store.
6. A captive portal usually does two things. It answers every DNS query with its own address, and it answers every plain HTTP request with a redirect to the login page.
7. That is why HTTPS sites fail with a certificate warning on a hotel network instead of showing the portal. The portal cannot forge a valid certificate, so the browser correctly refuses.
8. Modern operating systems work around this by deliberately making one plain HTTP request to a known address after joining a network, and checking whether the expected answer comes back.

9. If the answer is wrong, the system concludes a portal is present and opens the little login window automatically.

34.5.5 TECHNICAL - the engineer's version

1. RFC 3234, February 2002, "Middleboxes: Taxonomy and Issues", is the document that named and catalogued them. It lists twenty-two kinds and warns about exactly the failure modes we now live with.
2. The end-to-end argument comes from the 1984 paper "End-to-End Arguments in System Design" by Jerome Saltzer, David Reed and David Clark. Its claim is that functions belong at the endpoints unless the network can do them materially better.
3. Middleboxes cause **ossification**: the network becomes unable to accept new protocols because boxes drop what they do not recognize. Attempts to deploy new TCP options routinely fail for this reason.
4. This is a large part of why QUIC, standardized in RFC 9000 in May 2021, runs over UDP and encrypts almost its entire header. It is designed so middleboxes cannot parse it and therefore cannot ossify it.
5. Server Name Indication, the field carrying the site name in the clear, is defined in RFC 6066. Encrypted Client Hello is the proposed fix; it was still working through the IETF at the time of writing, and Cloudflare enabled it broadly in 2023. Treat it as emerging, not settled.
6. TLS interception at scale is a product category. Its risks are documented: a 2017 study by Durumeric and others, "The Security Impact of HTTPS Interception", found many interception products weakened the connection they proxied.
7. Two famous incidents are worth knowing. Lenovo shipped consumer laptops with Superfish adware carrying a private root certificate, discovered in February
2015. Kazakhstan required citizens to install a state root certificate in July 2019, and browser vendors blocked it.
8. Captive portal standards are recent. RFC 8910 and RFC 8908, both September 2020, define a DHCP and Router Advertisement option pointing at a machine readable portal API, so the operating system can be told rather than having to guess.
9. The probes each system uses are an implementation detail but worth knowing: Apple requests a tiny page from a Apple-operated host and expects the word "Success". Android requests a URL expecting HTTP status 204 with an empty body. Windows uses a similar text probe.
10. Filtering techniques, and how each one looks from the client, which is how you tell them apart:

Technique	Client symptom
DNS blocking	wrong or empty answer
IP or port blocking	timeout, no response
SNI-based reset	RST after ClientHello
HTTP redirect	portal page appears

11. Applying that table honestly to the reader's session: DNS returned a real address, so DNS blocking is ruled out. Nothing came back at all, and the failure was at the SYN, before any TLS ClientHello existed, so an SNI-based reset is ruled out too. What remains is address-level or port-level discarding, or a plain path fault. The evidence cannot separate those two.

34.5.6 WORDS - remember these

1. Middlebox — a device in the path that does more than forward — any on-path element that inspects, modifies or filters traffic.
2. Transparent proxy — an intercepting cache you did not configure — a proxy inserted by routing rather than by client settings.
3. Deep packet inspection — looking inside, not just at the label — examination of packet payloads or metadata to classify or control traffic.

4. TLS interception — legal, installed man-in-the-middle — decryption and re-encryption using a locally trusted root certificate.
5. Captive portal — the hotel login page — a network that intercepts the first request and redirects to an authentication page.
6. Ossification — the network refusing anything new — the inability to deploy new protocol behaviour because middleboxes drop unfamiliar traffic.

34.6 VPNs, what they do and what they do not

34.6.1 PLAIN - in simple words

1. A VPN, a virtual private network, does one mechanical thing: it wraps your traffic inside another connection to a server somewhere else, and lets that server send it onward.
2. Your packets go out looking like ordinary encrypted traffic to one address. Inside them are your real packets to their real destinations.
3. The VPN server unwraps them, sends them on with its own address, receives the replies, wraps them, and sends them back to you.
4. Two things change. First, your local network and your internet provider can no longer see where you are going. They only see the VPN server.
5. Second, the websites you visit see the VPN server's address, not yours, and your path to them now starts wherever that server is.
6. There are two main uses. One joins two whole offices into one network. The other lets a single laptop join a network it is not physically on.
7. The important honest point comes early: a VPN moves your trust. It does not remove it. Whoever runs the VPN can see what your provider used to see.
8. It is not anonymity. It does not stop tracking. It does not make you safe. It changes who is in a position to watch.

34.6.2 PLAIN - a picture in your head

1. Imagine you write letters, but you do not want your local post office to know who you write to.
2. So you put your addressed letter inside a second envelope and post it to a forwarding service in another city. The local post office only sees "letter to the forwarding service".
3. The service opens the outer envelope, posts your real letter with its own return address, receives the reply, and posts it back to you inside another outer envelope.
4. Your local post office learns nothing about your correspondents. Your correspondents learn nothing about your address.
5. The forwarding service, of course, learns everything. It sees every real address and, if the inner letter is not itself sealed, every word.
6. That is the trade in one paragraph. You have not become invisible. You have chosen a different party to be visible to.

Where this comparison breaks:

7. The inner letter is usually already sealed. Almost all web traffic is HTTPS, so the VPN provider sees which sites you visit and how much you transfer, but not the contents of pages.
8. And the forwarding service adds distance. If you are in India and the service is in Germany, every letter now travels to Germany first, which is why a VPN often makes things slower even when it works perfectly.

34.6.3 PLAIN – a worked example

1. Here is the reader's failing path, and the same request with a full-tunnel VPN to a server in Singapore.

```
Without a VPN:
laptop -> home router -> ISP core -> ISP edge ->
Microsoft Delhi -> Mumbai -> Pune -> nothing

With a full-tunnel VPN:
laptop -> home router -> ISP core -> ISP edge ->
VPN server in Singapore -> Microsoft Singapore edge ->
github.com  (works)
```

2. Everything after the VPN server is a different path across a different set of networks. If the device that dropped the packets sat on the old path and not the new one, the problem simply does not occur.
3. That is why "turn on a VPN" so often appears to fix an unexplained network fault. It is not magic. It is a different route.
4. Now look at what the tunnel looks like on the machine.

```
$ ifconfig utun4
utun4: flags=8051<UP,POINTOPOINT,RUNNING,MULTICAST> mtu 1420
       inet 10.7.0.6 --> 10.7.0.6 netmask 0xffffff00

$ netstat -rn | head -6
Destination    Gateway         Flags  Netif
0/1            10.7.0.5        UGSc   utun4
default        192.168.0.1     UGScg  en0
128.0/1        10.7.0.5        UGSc   utun4
203.0.113.77   192.168.0.1     UGHS   en0
```

5. Read those routes carefully, because they contain the cleverest trick in VPN software. The original default route through `192.168.0.1` is still there, untouched.
6. Instead, two routes were added: `0.0.0.0/1` and `128.0.0.0/1`. Together they cover the entire address space, and both are more specific than the default route, so both win.
7. The single exception is the fourth line: a direct route to the VPN server's own public address through the real gateway. Without that, the tunnel's own packets would try to go through the tunnel, which cannot work.
8. Note the MTU of 1420 rather than 1500. The wrapping costs bytes, so the inner packets must be smaller.

34.6.4 PLAIN – what is really happening inside

1. The VPN client creates a virtual network interface. On macOS these are called `utun`, on Linux `tun` or `wg0`, on Windows a virtual adapter.
2. To the operating system it looks like an ordinary network card, so ordinary routing rules can send traffic to it.
3. Anything the kernel routes into that interface is handed to the VPN software as a raw IP packet.
4. The software encrypts that packet, wraps it in a normal UDP or TCP packet addressed to the VPN server, and sends it out of the real interface.
5. So the same physical link now carries an outer packet, whose payload is a whole inner packet. This is **encapsulation**, and it is the same idea as putting one envelope inside another.
6. At the far end, the server decrypts, extracts the inner packet, and forwards it into the internet from its own address, keeping a translation record so it can return the reply to you.
7. Split tunnelling means only some routes point at the virtual interface. Work traffic goes through the tunnel, everything else goes out normally.
8. Full tunnelling means every route points at the virtual interface, so all traffic goes through the tunnel.

9. Split is faster and cheaper. Full is safer for a hostile local network, because nothing at all leaks to the local link.

34.6.5 TECHNICAL - the engineer's version

1. Deployment shapes: **site-to-site** joins two networks with a permanent gateway-to-gateway tunnel, usually IPsec, and users are unaware of it. **Remote access** connects one device into a network, and the user runs a client.
2. The protocol comparison, with the facts that decide real choices:

Protocol	Year	Transport	Note
PPTP	1999	TCP 1723, GRE	broken, do not use
L2TP/IPsec	1999-2001	UDP 500, 4500	slow, widely built in
IPsec IKEv2	2005-2014	UDP 500, 4500	best for roaming
OpenVPN	2001	UDP 1194 or TCP	flexible, userspace
WireGuard	2016-2020	UDP, one port	smallest and fastest

3. IPsec is a family, not one protocol. RFC 4301, December 2005, gives the architecture. ESP, RFC 4303, does the encryption. IKEv2, RFC 7296, October 2014, negotiates keys. NAT traversal wraps ESP in UDP port 4500 per RFC 3948. MOBIKE, RFC 4555, lets a tunnel survive an address change, which is why IKEv2 is the usual choice on phones.
4. OpenVPN was first released on 13 May 2001 by James Yonan. It runs in user space over OpenSSL or mbedTLS, defaults to UDP port 1194, and can be forced over TCP port 443 to look like ordinary web traffic, at the cost of TCP inside TCP, which behaves badly under loss.
5. WireGuard was written by Jason A. Donenfeld, first released publicly in 2016 and merged into the Linux kernel in version 5.6, released 29 March 2020. It is roughly 4,000 lines of kernel code, against hundreds of thousands for a full IPsec plus OpenVPN stack. Small code means fewer places for bugs and an auditable design.
6. WireGuard's speed comes from three choices. It runs in the kernel. It has no cipher negotiation at all: the algorithms are fixed at Curve25519, ChaCha20-Poly1305, BLAKE2s and HKDF, using the Noise protocol framework. And it is stateless in the sense that there is no session setup handshake per connection, only a periodic key exchange.
7. The trade-off is real and worth stating: with no negotiation, upgrading the cryptography means replacing the protocol version everywhere at once. The designers consider that a feature. Some enterprise architects consider it a problem. Both positions are defensible.
8. PPTP should never be used. RFC 2637, July 1999, is informational, and its authentication, MS-CHAPv2, was broken publicly by Moxie Marlinspike and David Hulton at Defcon in July 2012, reducing the effective work to a single DES key. Apple removed PPTP support in macOS Sierra 10.12 and iOS 10 in 2016.
9. L2TP alone, RFC 2661, August 1999, provides no encryption whatsoever. It is always paired with IPsec, per RFC 3193, and the pairing is often misconfigured.
10. About the reader's `utun` interfaces. `utun` is macOS's user tunnel kernel control interface, and a `utun` device is created by any process that asks the kernel for one. macOS itself creates several for its own features, so seeing `utun0` through `utun3` on a stock machine with no VPN is normal. Their presence is not evidence that a VPN was running.
11. To tell a real VPN from a system interface: a live tunnel has a routable address, at least one route pointing at it, and traffic counters that move. Check with `ifconfig utunN`, `netstat -rn`, `scutil --nc list` for configured VPN services, and `netstat -ibn` for byte counters.
12. What a commercial VPN genuinely protects against: a hostile or badly-run local network, such as a shared cafe or hotel WiFi; your access provider's visibility into which sites you reach; and geographic restrictions based on the apparent country of your address.

13. What it does not do, stated without softening. The provider now sees exactly what your provider used to see, and you have less legal recourse against them. It is not anonymity, because your account, your payment and your behaviour identify you. It does not stop cookies, browser fingerprinting, logins or advertising identifiers, which are all above the network layer. It does not protect a compromised device. And a "no logs" claim is a promise, not a mechanism.
14. Applied to the reader's fault: a full-tunnel VPN would probably have made the symptom disappear, because the path after the VPN server is entirely different. That is a workaround. It adds latency, costs money, moves trust to a third party, and leaves the broken path broken for everybody else on that provider. The fix is to report the fault with the evidence from Chapter 29.

34.6.6 WORDS - remember these

1. VPN — a private path built over a public one — an encrypted tunnel encapsulating IP traffic between a client or site and a gateway.
2. Encapsulation — a packet inside a packet — carrying a complete inner datagram as the payload of an outer one.
3. Split tunnelling — only some traffic goes through — a route configuration sending selected prefixes over the tunnel interface.
4. Full tunnelling — everything goes through — installing `0.0.0.0/1` and `128.0.0.0/1` so all traffic prefers the tunnel.
5. utun — the macOS virtual tunnel device — a kernel control interface providing point-to-point tunnel interfaces to user-space processes.
6. MOBIKE — a tunnel that survives changing networks — IKEv2 mobility and multihoming, RFC 4555.

34.7 Proxies, forward and reverse

34.7.1 PLAIN - in simple words

1. A proxy is a machine that makes a request on your behalf, or answers one on somebody else's behalf. Two jobs, two names, endlessly confused.
2. A **forward proxy** sits next to the clients. You send it your request, it fetches the page, it gives you the result.
3. A **reverse proxy** sits next to the servers. The public sends requests to it, and it decides which real server behind it should handle each one.
4. The memory rule, worth writing down: a forward proxy hides the client from the server. A reverse proxy hides the server from the client.
5. Forward proxies exist for filtering, caching, and control. Schools and companies use them.
6. Reverse proxies exist for spreading load, terminating encryption, caching, and routing by path or hostname. Almost every website has one.
7. You have used a reverse proxy every day of your life without knowing. You have probably only used a forward proxy at work or at school.
8. A proxy is not a VPN. A proxy handles particular connections for particular programs. A VPN handles all traffic for the whole machine.

34.7.2 PLAIN - a picture in your head

1. A forward proxy is a purchasing department. You are not allowed to buy directly. You submit a request, they buy the item, they hand it to you.
2. They can refuse certain purchases, keep a record, and reuse a bulk item they already have in stock instead of buying it again.
3. The supplier only ever deals with the purchasing department, and never learns which employee wanted the item.

4. A reverse proxy is a receptionist at a large company. Every visitor comes to the desk. The receptionist decides which department handles the visitor and walks them there.
5. Visitors never learn how many departments exist, where they sit, or that one of them was replaced last night.

Where this comparison breaks:

6. A receptionist handles one visitor at a time. A reverse proxy handles tens of thousands of connections at once and can send one visitor's request to two departments and merge the answers.
7. And the purchasing department knows what you bought. A forward proxy handling HTTPS traffic mostly does not, because it only opens a blind tunnel, unless it is also intercepting your encryption.

34.7.3 PLAIN - a worked example

1. The two shapes, drawn.

```
FORWARD PROXY (near the client)

 you -> proxy -> internet -> the site
        ^ you configured this
        ^ the site sees the proxy's address

REVERSE PROXY (near the server)

 you -> internet -> reverse proxy -> app server 1
                                  -> app server 2
                    ^ you never configured this
                    ^ you see one address and one name
```

2. Now a forward proxy in action for a plain HTTP request. The request line carries the whole address, which it never does otherwise.

```
GET http://example.com/page.html HTTP/1.1
Host: example.com
Proxy-Connection: keep-alive
```

3. For an encrypted request the proxy cannot do that, so the client asks for a tunnel instead, using a special method.

```
CONNECT example.com:443 HTTP/1.1
Host: example.com

HTTP/1.1 200 Connection established

(from here the proxy copies bytes blindly in both
 directions; the TLS handshake happens end to end)
```

4. The proxy therefore learns the hostname and port from the CONNECT line, and the number of bytes and the timing, but not the page contents.
5. Configuring one from the shell uses environment variables, which are a convention that most tools follow and a few ignore.

```
export http_proxy=http://proxy.corp.example:3128
export https_proxy=http://proxy.corp.example:3128
export no_proxy=localhost,127.0.0.1,.corp.example
curl -v https://example.com/     # now goes via CONNECT
curl --proxy socks5h://127.0.0.1:1080 https://example.com/
```

6. The `h` in `socks5h` matters: it means the proxy resolves the hostname, so your DNS queries do not leak to your local resolver.

34.7.4 PLAIN - what is really happening inside

1. With a forward proxy, your program opens a TCP connection to the proxy, not to the destination. There are two separate connections and the proxy joins them.
2. For plain HTTP the proxy fully understands the request and can cache, rewrite or block it.
3. For HTTPS the proxy is asked to become a pipe. After it answers "200 Connection established" it stops understanding anything and just copies bytes.
4. A SOCKS proxy is simpler and lower down. It does not understand HTTP at all. It just carries any TCP connection, and optionally UDP, to wherever you say.
5. That is why SOCKS works for SSH, databases, game protocols and anything else, while an HTTP proxy only works for HTTP and CONNECT tunnels.
6. With a reverse proxy, the public name and address belong to the proxy. It terminates the TCP connection and the encryption, reads the request, and decides.
7. It can then route on the hostname, on the path, on a header, or on a cookie, and it opens its own connection to a chosen backend server.
8. Because it terminates encryption, it can add caching, compression, request limiting and attack filtering in one place, for every backend behind it.
9. The cost is that the backend no longer sees the real client address, so the proxy must pass it on in a header, and the backend must be configured to trust that header only from the proxy.

34.7.5 TECHNICAL - the engineer's version

1. The distinction in one table:

Property	Forward proxy	Reverse proxy
Sits near	the client	the server
Configured by	the client	the site owner
Hides	the client	the servers
Typical software	Squid, tinyproxy	nginx, HAProxy

2. The CONNECT method is defined in RFC 9110, June 2022, section 9.3.6, and was earlier described in RFC 2817, May 2000. A proxy that permits CONNECT to arbitrary ports is an open relay and will be abused, so real proxies restrict it to 443 and a short list.
3. SOCKS version 4 was written by David Koblas around 1992. SOCKS5 is RFC 1928, March 1996, and adds authentication, IPv6, UDP association and, critically, remote name resolution via the domainname address type.
4. Proxy auto-configuration files are a JavaScript function named `FindProxyForURL(url, host)` returning a string such as `DIRECT` or `PROXY host:port`. Netscape introduced them in 1996. Web Proxy Auto-Discovery locates the file via DHCP option 252 or a DNS name, and has been a recurring security weakness because whoever answers first wins.
5. Proxy versus VPN, at the packet level. A proxy operates at layer 5 to 7: your application makes a new TCP connection to the proxy, and the proxy makes a separate one onward. Only applications that know about the proxy use it. ICMP, DNS and everything else bypass it. A VPN operates at layer 3: whole IP packets are encapsulated, so every protocol and every program is covered whether it knows or not.
6. Reverse proxy software, with real origins. `nginx` was written by Igor Sysoev and first released publicly on 4 October 2004, to solve the C10K problem of ten thousand concurrent connections. HAProxy was written by Willy Tarreau, first released in 2001, and is still the reference for layer 4 and layer 7 load balancing. Envoy was open-sourced by Lyft on 20 September 2016 and became the data plane of service

meshes such as Istio. Caddy, by Matt Holt, first released in 2015, was the first popular server to obtain and renew TLS certificates automatically.
7. Squid, the classic forward caching proxy, dates from 1996 and descends from the Harvest project.
8. Where reverse proxies sit in a real deployment, outermost first: a CDN edge, then a cloud load balancer, then an ingress reverse proxy such as nginx or Envoy, then the application. Four layers of proxy is normal, and each one can add its own timeout, its own header rewriting and its own failure mode.
9. Client address propagation uses `X-Forwarded-For`, a convention from Squid, or the standardized `Forwarded` header from RFC 7239, June 2014. Trusting these headers from anything other than your own proxy is a well-known vulnerability, because a client can simply send them.
10. Observation: `curl -v --proxy ...` shows the CONNECT exchange, `nginx -T` prints the full resolved configuration, and `echo "show stat" | socat /var/run/haproxy.sock stdio` dumps HAProxy counters.

34.7.6 WORDS - remember these

1. Forward proxy — fetches things for clients — a proxy configured by the client that hides clients from origin servers.
2. Reverse proxy — answers for servers — a proxy owned by the site that terminates client connections and dispatches to backends.
3. CONNECT — ask a proxy for a blind pipe — the HTTP method that establishes a TCP tunnel through a proxy, used for HTTPS.
4. SOCKS — a general-purpose connection relay — a layer 5 proxy protocol carrying arbitrary TCP and UDP, defined in RFC 1928.
5. PAC file — a script that picks the proxy — a JavaScript `FindProxyForURL` function evaluated by the client per request.
6. X-Forwarded-For — the original client address, passed along — a convention header, standardized as `Forwarded` in RFC 7239.

34.8 Tunnelling in general

34.8.1 PLAIN - in simple words

1. Tunnelling means putting one packet inside another packet.
2. The outer packet is addressed to a machine that knows how to unwrap it. The inner packet is addressed to where you actually wanted to go.
3. The network in between reads only the outer packet. It never looks at the inner one and does not need to understand it.
4. That is the whole trick. Everything called a tunnel is a version of it.
5. There are three honest reasons to tunnel. To carry something the network refuses to carry. To hide what you are carrying. To make two far-apart networks behave like one.
6. Because the outer packet needs its own header, a tunnel always makes packets bigger, so every tunnel eats a little of your usable space.
7. A tunnel is not automatically private. Wrapping and encrypting are different jobs. Some tunnels encrypt, many do not.
8. SSH gives you three ready-made tunnels for free. They are among the most useful tools a developer owns and almost nobody is taught them.
9. There are also services whose only job is to give a machine behind a home router a public address, by having that machine dial out first.

34.8.2 PLAIN – a picture in your head

1. You want to send a letter into a building that does not accept post from outside.
2. So you seal your letter, write your colleague's internal room number on it, and put it inside a bigger envelope addressed to the reception desk.
3. The postal service only reads the outer envelope. It delivers to reception.
4. Reception opens the outer envelope, sees an internal room number, and passes the inner letter along the internal mail system.
5. Reception is the tunnel endpoint. The outer envelope is the encapsulation. The inner letter is your real packet.
6. The outer envelope also weighs something, so a fixed weight limit now holds slightly less of your actual letter.

Where this comparison breaks:

7. The post office does not silently destroy an envelope that is one gram too heavy. Networks do exactly that when a packet is too big and the message saying so is filtered. You get silence, not a bounce.
8. And reception can read your inner letter unless you sealed it. Wrapping alone gives you no privacy at all. Only encryption does.

34.8.3 PLAIN – a worked example

1. SSH has three forwarding modes. Learn the letters and you own them for life.
2. `-L` is local forwarding. It opens a port on your machine and sends anything arriving there out through the SSH server.

```
# reach a database that only the bastion host can see
ssh -N -L 5433:db.internal:5432 user@bastion.example.com
# now, in another shell, on your own laptop:
psql -h 127.0.0.1 -p 5433 -U app appdb
```

3. Read `-L 5433:db.internal:5432` as: listen on my port 5433, and from the SSH server open a connection to `db.internal` port 5432.
4. `-R` is remote forwarding. It is the mirror image. It opens a port on the SSH server and sends anything arriving there back to you.

```
# let a colleague see the site running on your laptop
ssh -N -R 9000:localhost:3000 user@public.example.com
# they open port 9000 on public.example.com and reach
# port 3000 on your laptop
```

5. By default that remote port only listens on the server's own loopback. To let other people reach it, the server needs `GatewayPorts yes` in its `sshd_config`. That is a deliberate safety default.
6. `-D` is dynamic forwarding. It turns your SSH connection into a SOCKS proxy, so any program can send any connection through it.

```
ssh -N -D 1080 user@bastion.example.com
curl --proxy socks5h://127.0.0.1:1080 https://intranet.example/
```

7. `-N` means "do not run a command on the far end, just forward". `-f` puts it in the background. Together they are the usual pair.
8. The three modes drawn, because the arrows are the whole point.

```
-L LOCAL FORWARD   (you open the door on your side)

  your app -> 127.0.0.1:5433 -[ssh]-> bastion -> db:5432

-R REMOTE FORWARD  (you open the door on their side)
```

```
  visitor -> public:9000 -[ssh]-> your laptop -> :3000

-D DYNAMIC         (a SOCKS proxy, any destination)

  any app -> 127.0.0.1:1080 -[ssh]-> bastion -> anywhere
```

9. Now the outside-in version. `ngrok` and Cloudflare Tunnel do the `-R` trick as a product, with a public name and a certificate.

```
ngrok http 3000
cloudflared tunnel --url http://localhost:3000
```

10. Both print a public hostname. Both work from behind a home router with no port forwarding, because the agent dialled out and the router allowed that outbound connection, exactly as it allows any web request.

34.8.4 PLAIN - what is really happening inside

1. SSH forwarding is not really packet-in-packet. It is connection-in-connection, and the difference matters.
2. Your `ssh` client opens a normal listening socket on your machine, on the port you asked for.
3. When a program connects to that port, the client opens a new logical channel inside the one existing SSH connection and says "please connect to this host and port".
4. The SSH server makes a brand new TCP connection from itself to the target, and then copies bytes between that connection and the channel.
5. So the target sees a connection coming from the SSH server, not from you. Its logs show the bastion's address. Its firewall rules must allow the bastion.
6. Many channels share one SSH connection, so many forwards cost one TCP connection and one handshake.
7. A real network tunnel such as VXLAN or GRE works one level lower. It takes a whole packet, headers and all, and puts it in the payload of a new packet.
8. The receiving end strips the outer header and injects the inner packet as if it had just arrived on a wire. Nothing above it knows a tunnel happened.
9. That is why a layer 3 tunnel carries every protocol, and an SSH forward carries only the TCP connections you named.
10. In both cases the outer header takes space. If the original packet was already full size, the wrapped packet is now too big for the path, and it must be fragmented, refused, or dropped.

34.8.5 TECHNICAL - the engineer's version

1. Encapsulations you will meet, with their real identifiers and overhead over IPv4.

Tunnel	Carried as	Overhead
IP-in-IP, RFC 2003	IP protocol 4	20 bytes
GRE, RFC 2784	IP protocol 47	24 bytes
IPsec ESP tunnel	IP protocol 50	50 to 73 bytes
VXLAN, RFC 7348	UDP port 4789	50 bytes
Geneve, RFC 8926	UDP port 6081	50 bytes plus
WireGuard	UDP, chosen port	60 bytes

2. GRE, Generic Routing Encapsulation, is RFC 2784, March 2000, updated by RFC 2890, and replaced the earlier RFC 1701 of October 1994. It has no encryption and no authentication at all. It is a plain wrapper, still used between routers and by DDoS scrubbing providers to return cleaned traffic.

3. VXLAN, Virtual eXtensible LAN, is RFC 7348, August 2014. It puts a whole Ethernet frame inside UDP. Its 24-bit VXLAN Network Identifier gives 16,777,216 segments, against 4,094 usable VLANs, which is the reason it exists. Cloud overlays and Kubernetes network plugins use it constantly.
4. Geneve, RFC 8926, November 2020, is the newer extensible version, and is what the AWS Gateway Load Balancer speaks.
5. MTU arithmetic, the part that bites. On a 1500-byte Ethernet path, a VXLAN tunnel leaves 1450 bytes for the inner frame. WireGuard's `wg-quick` sets the tunnel interface MTU to 1420 by default for the same reason.
6. When encapsulation shrinks the path and the ICMP "fragmentation needed" message is filtered, Path MTU Discovery fails silently and you get the classic signature: the handshake works, small responses arrive, and the first large response hangs forever. The fix is usually TCP MSS clamping, for example `iptables -t mangle -A FORWARD -p tcp --syn -j TCPMSS --clamp-mss-to-pmtu`.
7. SSH port forwarding is defined in RFC 4254, January 2006, the SSH Connection Protocol, as channels of type `direct-tcpip` for `-L` and `forwarded-tcpip` for `-R`. SOCKS5 dynamic forwarding is RFC 1928, March 1996. OpenSSH itself was forked from Tatu Ylonen's SSH 1.2.12 by the OpenBSD project and first released in December 1999.
8. Do not tunnel TCP inside TCP if you can avoid it. Two retransmission timers stack, and a lost packet triggers both, which is why `ssh -w` and TCP-based VPNs collapse on lossy links. UDP-based tunnels do not have this problem.
9. Outbound-dialled tunnel services, with real dates. ngrok was created by Alan Shreve and released in 2013. Cloudflare launched Argo Tunnel in 2018 and renamed it Cloudflare Tunnel in 2021; its agent is `cloudflared` and it connects outbound over QUIC or HTTP/2. Tailscale added Funnel in 2022.
10. The security consequence is blunt and worth stating once. A public tunnel to `localhost` exposes that service to the entire internet, including automated scanners that find new hostnames within minutes. Put authentication in front of it before you start it, not after.
11. Observation commands: `ssh -v` prints each channel as it opens, `ip link show type vxlan` and `bridge fdb show` inspect a VXLAN interface on Linux, `tcpdump -ni any udp port 4789` shows the encapsulated frames, and `ip route get 10.0.0.5` tells you whether a packet will take the tunnel.

34.8.6 WORDS - remember these

1. Tunnel — a packet carried inside another packet — encapsulation of one protocol's payload within another protocol for transit.
2. Encapsulation — the wrapping itself — prepending an outer header so intermediate routers forward on the outer address only.
3. Local forward — a door on your side — `ssh -L`, a listening socket on the client relayed to a target reached from the server.
4. Remote forward — a door on their side — `ssh -R`, a listening socket on the server relayed back to a target reached from the client.
5. Dynamic forward — a proxy, not a fixed pair — `ssh -D`, a local SOCKS5 listener whose destinations are chosen per connection.
6. VXLAN — Ethernet inside UDP — RFC 7348 overlay with a 24-bit VNI and 50 bytes of overhead, UDP port 4789.
7. GRE — a plain wrapper with no encryption — RFC 2784, IP protocol 47, 24 bytes of overhead.
8. MSS clamping — telling both ends to use smaller pieces — rewriting the TCP maximum segment size option to fit the tunnel's path MTU.

34.9 CDNs and anycast in depth

34.9.1 PLAIN - in simple words

1. A CDN, a content delivery network, is a large set of computers placed close to users, holding copies of somebody else's files.
2. The point is distance. A file served from a machine 40 kilometres away arrives far sooner than the same file from 13,000 kilometres away, and no amount of money changes that.
3. The site that owns the content is called the **origin**. The CDN machines are called **edges** or points of presence.
4. A CDN can copy anything that is the same for everybody: images, style sheets, scripts, fonts, video pieces, downloads.
5. A CDN cannot copy things that are different for each person, such as your account page or your shopping basket, unless the site takes special care.
6. There are two ways to fill a CDN. Push, where you upload files to it in advance. Pull, where it fetches a file from the origin the first time anyone asks, and keeps it.
7. Nearly everything today is pull, because it needs no work from you.
8. **Anycast** is a separate idea that CDNs also use. It means one address exists in many places at once, and the network delivers you to whichever copy is closest by its own reckoning.
9. Anycast and DNS steering both send different users to different places, and people mix them up constantly. One works in the routing system, the other in the naming system. We will separate them properly at the end.

34.9.2 PLAIN - a picture in your head

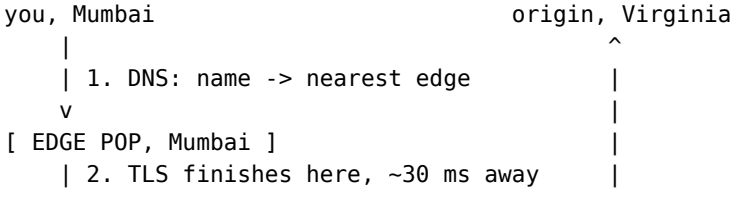
1. Think of a popular book and a national library system.
2. The origin is the one library that owns the original manuscript.
3. Rather than making every reader travel to that one building, the system prints copies and puts them in every town branch.
4. The first reader in your town asks for the book, the branch does not have it, so the branch orders one copy from the central library and keeps it on the shelf.
5. Every reader after that is served in two minutes from the local shelf. That is a cache hit.
6. The branch is told how long the copy stays valid. After that it must check with the central library before lending it again.
7. Now the anycast part. Imagine every branch has the same street address painted on the door, and the road signs in each town point to the nearest one. You always drive to "1 Library Road" and always arrive somewhere local.

Where this comparison breaks:

8. Books do not change under you. Web content does, and a stale copy on a shelf can be actively wrong, which is why purging and versioned file names exist.
9. Road signs do not change while you are driving. Internet routes do, and if they change mid-journey an anycast connection can arrive at a branch that has never heard of you and be turned away.
10. And a library branch does not run your code. A modern edge does, which is the part of this comparison that has aged the worst.

34.9.3 PLAIN - a worked example

1. One request for one image, walked end to end, with a cold cache.



```
     | 3. build cache key, look it up        |
     |      HIT  -> send bytes, done         |
     |      MISS -> ask the shield           |
     v                                       |
 [ SHIELD POP, one per origin ] --- 4. ------+
     5. shield stores it, sends it to edge
     6. edge stores it, sends it to you
```

2. Step 3 needs a cache key. By default it is built like this.

```
key = scheme + host + path + query string
    = https + img.example.com + /logo.avif + ?v=7
```

3. Step 4 only happens on a miss. With a shield in front of the origin, 200 edges missing the same file cause one origin request, not 200.
4. What the origin sends back, and what each header does.

```
HTTP/1.1 200 OK
Content-Type: image/avif
Cache-Control: public, s-maxage=600, max-age=60
ETag: "9f2a1c"
Vary: Accept-Encoding
```

5. `max-age=60` tells your browser to keep it for 60 seconds. `s-maxage=600` tells shared caches such as the CDN to keep it for 600 seconds. Shared caches prefer `s-maxage` when both are present.
6. `Vary: Accept-Encoding` adds that request header to the cache key, so the compressed and uncompressed versions are stored separately.
7. What the edge sends you, with two extra headers of its own.

```
HTTP/1.1 200 OK
Age: 143
X-Cache: HIT
ETag: "9f2a1c"
```

8. `Age: 143` means this copy has been in caches for 143 seconds, so it has 457 seconds of its 600 left. `X-Cache: HIT` is a convention, not a standard; every CDN spells it differently.
9. Now the timing, using round-trip times we will justify in section 34.10.

Case	Where served	Time to first byte
Cache hit, Mumbai edge	30 ms away	about 45 ms
Cache miss, shield hit	30 ms plus	about 250 ms
Cache miss to origin	Virginia	about 600 ms

10. That table is the entire business case for CDNs in three rows.

34.9.4 PLAIN – what is really happening inside

1. The name you type does not resolve to the origin. It resolves, one way or another, to a CDN address near you.
2. There are only two mechanisms for that, and this is the crux of the whole section.
3. Mechanism one, DNS steering. The authoritative name server looks at who is asking and returns a different address to different askers. Different users get different addresses.
4. Mechanism two, anycast. Every user gets the same address, and the routing system itself carries each user to a different physical machine.
5. Most large CDNs use both: anycast for the edge addresses, and DNS to hand out which anycast address family or which service you land in.

6. Once you reach the edge, the edge terminates your TCP connection and your encryption. That is why the handshake feels fast: it finished 30 ms away, not 200 ms away.
7. The edge computes the cache key, looks in memory, then on local flash. A hit is answered immediately.
8. A miss is not sent straight to the origin. It goes to a parent or shield node, and identical concurrent misses are collapsed into one upstream request so the origin is never stampeded.
9. The connection from edge to origin is usually already open, already warmed up, and often runs over the CDN's own private network rather than the public internet.
10. Stale content is handled by time, not by magic. When the timer expires the edge revalidates with a conditional request, and a `304 Not Modified` reply costs one small round trip instead of a whole file.
11. If you need something gone before its timer expires, you purge it, which is a message pushed out to every edge holding it.
12. Edge compute means the edge can also run small programs of yours before or after the cache lookup, so decisions that used to need the origin now happen 30 ms from the user.

34.9.5 TECHNICAL – the engineer's version

1. Cacheability rules, from RFC 9111, June 2022, which replaced RFC 7234. A shared cache may store a response if the method is cacheable, the status is understood, and no directive forbids it.
2. The directives that decide it, and what they really mean.

Directive	Effect on a CDN
`public`	may be stored by shared caches
`private`	browser only, never the CDN
`no-store`	nobody stores it at all
`no-cache`	store, but always revalidate
`s-maxage=N`	lifetime for shared caches
`immutable`	never revalidate before expiry

3. `stale-while-revalidate` and `stale-if-error` come from RFC 5861, May 2010. The first serves the old copy while fetching a new one in the background. The second serves the old copy when the origin is down, and it has saved more outages than any dashboard.
4. Validators: `ETag` with `If-None-Match`, and `Last-Modified` with `If-Modified-Since`, producing `304 Not Modified` with no body.
5. Cache key control is where real hit rates are won. Strip tracking query parameters, normalize the order of the remaining ones, and never send `Vary: User-Agent`, which multiplies your object count by thousands.
6. Purging, in increasing order of danger: by exact URL, by surrogate key or tag, by prefix, and purge-everything. Fastly calls the tag header `Surrogate-Key`, Cloudflare calls it `Cache-Tag`, Akamai uses `Edge-Control` and its own cache tags. Purge-everything on a busy site sends every edge to the origin at once and can take the origin down.
7. The better pattern is to never purge. Put a content hash in the file name, serve it with `max-age=31536000, immutable`, and change the name when the content changes. One year is the maximum value RFC 9111 encourages.
8. Edge compute, with real dates. Cloudflare Workers were announced on 29 September 2017, running V8 isolates rather than containers. AWS Lambda@Edge became generally available in July 2017 and CloudFront Functions in May 2021. Fastly's Compute product, built on WebAssembly, reached general availability in 2020.
9. Deployment shapes differ, and the industry genuinely disagrees about which is better. These are company-published figures as of 2026, which is to say marketing claims rather than audited counts, and they change constantly.

CDN	Shape	Published claim
Akamai	many small POPs	around 4,000 locations
Cloudflare	mid-size POPs	over 300 cities
Fastly	few large POPs	roughly 100 POPs

10. The argument: many small POPs sit closer to users; few large POPs hold more objects each, so they have higher hit rates and better shielding. Both camps are right about their own metric.
11. History: Akamai was founded in 1998 by Tom Leighton and Daniel Lewin at MIT, out of work on consistent hashing, and launched service in 1999. Cloudflare was founded in 2009 by Matthew Prince, Lee Holloway and Michelle Zatlyn and launched publicly at TechCrunch Disrupt in September 2010. Fastly was founded in 2011 by Artur Bergman.
12. Now anycast, properly. Anycast was described in RFC 1546, November 1993, by Craig Partridge, Trevor Mendez and Walter Milliken. Operational guidance is RFC 4786, December 2006, which is BCP 126.
13. The mechanism is entirely BGP. The same prefix, say a `/24`, is announced from many autonomous systems or many locations of one autonomous system. Every router in the world independently picks one best path to that prefix using its usual decision process: local preference, then AS path length, then origin, then MED, then eBGP over iBGP, then IGP cost.
14. Therefore "nearest" in anycast means nearest in BGP policy terms, not in kilometres. A site two countries away with a shorter AS path will win over a site in your city that your ISP reaches through three transit providers.
15. Why UDP and DNS love anycast: a DNS query is one packet out, one packet back, with no state at the server. If the next query lands at a different site, nothing is lost. The 13 root server identities are served this way from more than 1,900 instances worldwide, a figure that changes monthly.
16. Why TCP needs care: every packet of a flow must reach the same site, because only that site holds the transmission control block. A packet arriving at a site with no matching socket gets a RST and the connection dies.
17. In practice it works because routes are stable for minutes to hours, and because equal-cost multipath hashing inside a network is per-flow, keyed on the five tuple, so all packets of one connection follow one path.
18. A session sticks to one site for exactly as long as the routing decision at every hop stays the same. It is not pinned, tracked or guaranteed. It is merely stable, which is a weaker and more honest word.
19. When routing changes mid-connection, the flow is delivered to a different POP, which resets it. The user sees a broken download or a stalled upload. Short connections almost never notice. Long ones do.
20. Mitigations used in production: keep announcements stable and drain sites gracefully rather than withdrawing routes instantly, use consistent hashing in the load balancers behind the anycast address, and for QUIC use the connection ID, which lets a new front end recognize an existing connection and route it correctly, as specified in RFC 9000, May 2021.
21. A common hybrid is anycast for the first contact and a unicast address for the long transfer, which gives fast site selection and stable long flows.
22. Anycast inherits BGP's weaknesses. A route hijack sends your users to somebody else's machine. Real cases: Pakistan Telecom's accidental global hijack of YouTube on 24 February 2008, and the deliberate hijack of Amazon Route 53 address space on 24 April 2018 used to steal cryptocurrency from MyEtherWallet users. RPKI route origin validation exists to reduce this.
23. The final separation, which is the most confused pair in networking.

Question	Anycast	DNS steering
Which layer	routing, BGP	naming, DNS
Addresses	one, everywhere	different per user

Question	Anycast	DNS steering
Who decides	the internet's routers	the authoritative server
Failover time	seconds, BGP reconverges	one TTL, 30 to 300 s

24. Two more differences that matter in an incident. DNS steering can be overridden by a client that ignores TTLs or pins an address, and there are many such clients. Anycast cannot be overridden by the client at all.
25. And DNS steering sees the resolver, not the user. The reader's own session used `1.1.1.1`, Cloudflare's public resolver, rather than the ISP's. With DNS steering that can put you in the wrong region, unless the resolver forwards a truncated client subnet using EDNS Client Subnet, RFC 7871, May
2016. Cloudflare's resolver deliberately does not send client subnet by default, for privacy; Google's public resolver does.
26. In the reader's case the answer was still local: `github.com` resolved to `20.207.73.82`, an address in a Microsoft-owned range serving India, and the traceroute entered Microsoft's network at a Delhi site named `ae66-0.del01-96cbe-1b.ntwk.msn.net`. The evidence supports local steering working correctly. It does not prove which mechanism produced it.
27. Observation: `dig +short github.com` from two networks shows steering, `dig CHAOS TXT id.server @1.1.1.1` or `hostname.bind` reveals which anycast instance answered you, and `curl -sI` plus a look at `Age`, `X-Cache` or `CF-Cache-Status` tells you whether you got a cached copy.

34.9.6 WORDS - remember these

1. CDN — machines near you holding copies of a site's files — a distributed caching and delivery network fronting an origin.
2. Origin — the real server that owns the content — the authoritative upstream a CDN fetches from on a miss.
3. Pull versus push — fetch on first request versus upload in advance — origin pull caching versus pre-positioned distribution.
4. Cache key — the label a cached copy is filed under — normally scheme, host, path and query, extended by `Vary` headers.
5. Origin shield — one CDN node that speaks to the origin — a mid-tier cache collapsing edge misses into a single upstream request.
6. Purge — delete a copy before it expires — an invalidation pushed to every edge, by URL, tag or prefix.
7. Edge compute — your code running in the CDN — isolate or WebAssembly runtime executing per request at the point of presence.
8. Anycast — one address in many places — the same prefix announced by BGP from multiple sites, each router choosing its best path.
9. DNS steering — a different answer per asker — authoritative DNS returning addresses chosen by resolver location, latency or health.
10. EDNS Client Subnet — telling the server roughly where the user is — RFC 7871 option carrying a truncated client prefix to the authoritative server.

34.10 Latency versus bandwidth

34.10.1 PLAIN - in simple words

1. Latency is the waiting time before anything arrives. Bandwidth is how much arrives per second once it starts.
2. They are different quantities with different units, and they have almost nothing to do with each other.
3. Latency is measured in milliseconds. Bandwidth is measured in megabits per second. You cannot convert one into the other.

4. Your internet plan sells you bandwidth. Your experience of a slow website is almost always latency.
5. Latency has a hard floor set by physics. Nothing travels faster than light, and light in glass is slower than light in air.
6. Bandwidth has no such floor. You can always lay another cable, or send more colours of light down the same one.
7. So bandwidth is an engineering and money problem, and latency, past a point, is a geography problem.
8. The single most useful sentence in this chapter: **you can buy more bandwidth, you cannot buy a shorter distance.**
9. A page that makes 60 small requests to a server 200 milliseconds away is slow no matter how fast your connection is, because it spends its life waiting, not transferring.

34.10.2 PLAIN – a picture in your head

1. Think of a road between two cities.
2. Latency is how long the road is, divided by the speed limit. It is the time for one car to make the trip.
3. Bandwidth is how many lanes the road has. It is how many cars can be moving at once.
4. Adding lanes does not shorten the road. A ten-lane motorway and a two-lane road of the same length take one car the same time to drive.
5. If you need to move ten thousand cars, lanes are what you want. If you need to move one car and get an answer back, length is what you want.
6. That is the whole difference, and most complaints about "slow internet" are complaints about the length of the road.

Where this comparison breaks:

7. Cars do not wait for permission. Data does. A sender may only put a limited amount on the road before it must wait for confirmation from the far end, so the length of the road actually limits how many lanes you can use.
8. On a real motorway, adding cars makes everyone slower. On a fibre link, the cable's capacity is fixed, and what actually slows down is the queue at the entrance, which is a different mechanism with a name: bufferbloat.
9. And the road is not straight. Cables follow coasts, avoid mountains and share trenches, so the real distance is commonly 1.3 to 1.6 times the map distance.

34.10.3 PLAIN – a worked example

1. Start with physics. Light in vacuum travels at 299,792 kilometres per second.
2. Light in optical fibre travels slower, because glass has a refractive index of about 1.47. That gives roughly 204,000 kilometres per second.
3. Round it to 200,000 kilometres per second, which is 200 kilometres per millisecond, and you get a rule of thumb worth memorizing.
4. **Ideal round trip in milliseconds is roughly the one-way distance in kilometres divided by 100.** The 100 rather than 200 is because a round trip covers the distance twice.
5. Mumbai to Ashburn, Virginia, where a large part of the United States east coast cloud sits, is about 12,850 kilometres in a straight line.
6. So the ideal round trip is about 12,850 divided by 100, which is about 128 milliseconds. That is with a perfectly straight cable and zero equipment.
7. Real measurements from an Indian home connection to that region are around 185 to 200 milliseconds. The extra 60 or so milliseconds is cable routing, plus every router, queue and last-mile hop on the way.
8. No purchase, no protocol and no optimization will ever take that number below 128 milliseconds, short of digging a straighter trench.
9. Now the second calculation, the one that explains slow pages. Suppose a 100 megabit per second line and a 200 millisecond round trip.

10. How much data can be in flight at once? That is the bandwidth-delay product.

```
BDP = bandwidth x round-trip time
    = 100,000,000 bits/s x 0.200 s
    = 20,000,000 bits
    = 2,500,000 bytes
    = about 2.5 megabytes
```

11. To use the full 100 megabits per second, the sender must be allowed to have 2.5 megabytes unacknowledged at all times.
12. The original TCP window field is 16 bits, so without the window scaling option the largest window is 65,535 bytes.
13. That caps throughput at 65,535 bytes divided by 0.2 seconds, which is about 328 kilobytes per second, or about 2.6 megabits per second.
14. On a 100 megabit line. That is 2.6 percent of what you pay for, and the cause is entirely the round trip.
15. Finally, why more bandwidth does not help a page. Take a 100 kilobyte response over that same 200 millisecond link.

Step	At 10 Mbit/s	At 100 Mbit/s
DNS, TCP, TLS, request	800 ms	800 ms
Sending 100 KB	80 ms	8 ms
Total	880 ms	808 ms

16. Ten times the bandwidth bought 8 percent. Halving the round trip to 100 milliseconds would have given 480 milliseconds, a 45 percent saving.

34.10.4 PLAIN - what is really happening inside

1. A round trip is not one thing. It is four things added together, and only one of them is physics.
2. Propagation delay: the time for the signal to travel the distance. Fixed by the speed of light in the medium. You cannot improve it.
3. Serialization delay: the time to push the bits of a packet onto the wire. A 1500-byte packet takes 1.2 milliseconds at 10 megabits per second and 12 microseconds at 1 gigabit per second.
4. Queuing delay: time spent waiting in a buffer behind other packets. This is the one that varies wildly and the one that badly sized buffers make worse.
5. Processing delay: the time each router takes to look up and forward. On modern hardware this is microseconds.
6. On a long path, propagation dominates. On a congested last mile, queuing dominates. Both are latency, and neither is fixed by more bandwidth.
7. Now the feedback loop. TCP does not send everything at once. It starts small and grows, because it does not know what the path can take.
8. It starts with an initial window of about ten packets, roughly 14 kilobytes, and it doubles that each round trip while nothing is lost.
9. So a 100 kilobyte file takes about three round trips of growth, whatever the line speed is. At 200 milliseconds each, that is 600 milliseconds of waiting, not transferring.
10. Every extra round trip in your protocol costs the full latency, every time. DNS is one. The TCP handshake is one. Modern encryption is one more. A redirect is another full set.
11. That is why page load time is latency-bound. A typical page is not one transfer. It is dozens of small transfers, each one paying the round trip again, and many of them cannot start until an earlier one finished.
12. Reducing the number of round trips is therefore the single highest-value performance work there is, and it is usually free.

34.10.5 TECHNICAL - the engineer's version

1. Typical round-trip times from a home broadband connection in India, against the theoretical floor. Measured values are approximate, vary by provider and by time of day, and are given as ranges for that reason.

From India to	Distance	Ideal RTT	Typical RTT
Chennai	1,030 km	10 ms	25 to 35 ms
Bahrain, Gulf	2,420 km	24 ms	35 to 45 ms
Singapore	3,900 km	38 ms	55 to 65 ms
Frankfurt	6,560 km	64 ms	105 to 120 ms
Tokyo	6,720 km	66 ms	120 to 135 ms
London	7,190 km	70 ms	110 to 125 ms
Sydney	10,160 km	99 ms	150 to 165 ms
Oregon, US West	12,690 km	124 ms	215 to 235 ms
Virginia, US East	12,850 km	126 ms	185 to 200 ms
Sao Paulo	13,770 km	135 ms	320 to 360 ms

2. Distances are great-circle from Mumbai. Ideal round trip assumes 204,000 kilometres per second in fibre, which is the speed of light divided by a refractive index of 1.4675, a normal figure for standard single-mode fibre.
3. Note Tokyo and Oregon. Both are further by cable than by map, because the cable routes run through Singapore and around the Pacific rather than over central Asia. The ratio of real to ideal is the honest measure of a route.
4. Bandwidth-delay product, formally: BDP in bytes equals bandwidth in bits per second times round-trip time in seconds, divided by eight. It is the amount of data that must be in flight to keep the pipe full.
5. TCP window scaling is RFC 7323, September 2014, which replaced RFC 1323 of May 1992. The scale factor shifts the 16-bit window left by up to 14 bits, giving a maximum window of 1 gigabyte. It is negotiated in the SYN, so both ends must offer it, and a middlebox that strips the option silently caps your throughput.
6. Initial congestion window of 10 segments is RFC 6928, April 2013. Before it, the standard was 2 to 4 segments, and a small page took twice as many round trips.
7. Loss matters more than bandwidth on long paths. The Mathis equation approximates a single TCP flow's rate as MSS divided by RTT, times one over the square root of the loss probability.
8. Worked: 1,460-byte segments, 200 millisecond round trip, 0.1 percent loss. That gives about 1.85 megabits per second for one connection, on a line of any speed. This is why long-distance transfers use many parallel connections or a loss-tolerant protocol.
9. Queuing delay and bufferbloat: the term was popularized by Jim Gettys in 2010 and 2011. Oversized, unmanaged buffers in home routers add hundreds of milliseconds of latency under load. The modern fixes are CoDel, RFC 8289, and fq_codel, RFC 8290, both January 2018, plus BBR congestion control from Google in 2016.
10. Protocol round-trip costs, cold start, one round trip written as 1 RTT.

Step	Cost	Note
DNS lookup	1 RTT	0 if cached
TCP handshake	1 RTT	0 with TCP Fast Open
TLS 1.2	2 RTT	RFC 5246, 2008
TLS 1.3	1 RTT	RFC 8446, 2018
QUIC, new	1 RTT	RFC 9000, 2021

Step	Cost	Note
QUIC, resumed	0 RTT	replay risk

11. Page weight for scale: the HTTP Archive's public data has put the median page in the mid-2020s at roughly 2.5 megabytes across about 70 requests. Approximate, and it drifts upward every year.
12. Measurement tools, and what each actually measures. `ping` gives ICMP round trip, which some networks deprioritize. `mtr` and `traceroute` give per-hop round trip, which is the time to that hop and back, not a segment time. `iperf3` measures achievable throughput, not latency. `curl -w` measures the phases of a real request, which is what section 34.13 uses.
13. One caution from the reader's own session. Round-trip time to a hop tells you nothing about whether traffic passes through it. The reader's trace reached `ae106-0.rwa04.pnq20.ntwk.msn.net` at hop 12 and then went silent, and the silence was not a latency problem at all.

34.10.6 WORDS - remember these

1. Latency — the wait before anything arrives — one-way or round-trip delay, measured in milliseconds.
2. Round-trip time — there and back — RTT, the interval from sending a packet to receiving its acknowledgement.
3. Bandwidth — how much per second — capacity in bits per second, distinct from achieved throughput.
4. Throughput — what you actually got — measured delivery rate, always less than or equal to bandwidth.
5. Propagation delay — travel time over the distance — distance divided by the signal velocity, about 200,000 km/s in fibre.
6. Serialization delay — time to push bits onto the wire — packet size divided by link rate.
7. Bandwidth-delay product — how much can be in flight — bandwidth times RTT, the window size needed to fill a path.
8. Window scaling — permission to have more in flight — RFC 7323 TCP option raising the maximum window to 1 GiB.
9. Bufferbloat — queues so long they add delay — excessive unmanaged buffering, addressed by CoDel and fq_codel.

34.11 IPv6 properly

34.11.1 PLAIN - in simple words

1. Every device on the internet needs an address. The old kind, IPv4, is four numbers such as `20.207.73.82`, and there are about 4.3 billion of them.
2. The world has far more than 4.3 billion connected devices, so we ran out. That was known to be coming in the early 1990s.
3. IPv6 is the replacement. Its addresses are four times longer, written in hexadecimal and separated by colons, like `2001:db8::1`.
4. The number of IPv6 addresses is about 340 undecillion, a 3 followed by 38 digits. It is not a bigger pool. It is a pool of a completely different kind.
5. Because addresses are no longer scarce, every device can have its own public address, and the address-sharing trick called NAT becomes unnecessary.
6. IPv6 is not a newer version of IPv4 that IPv4 machines can talk to. It is a separate, parallel internet. A machine with only IPv6 cannot reach a machine with only IPv4.
7. That single fact explains why adoption took thirty years: everyone had to run both, which is more work, not less, until almost everyone has moved.
8. Most machines therefore run **dual stack**: both address families at once, choosing per connection.
9. The reader's own session printed `IPv6: (none)`. We will say exactly what that means, and exactly what it did and did not cause.

34.11.2 PLAIN – a picture in your head

1. IPv4 is a country that issued eight-digit phone numbers, then grew, and ran out of numbers.
2. Its answer was to give one real number to a whole building and put a switchboard in the lobby. Everyone inside dials out through the switchboard, and nobody outside can dial a specific flat directly. That is NAT.
3. IPv6 is a new numbering plan with numbers so long that every room, every appliance and every future appliance can have its own.
4. So the switchboard is no longer needed. Every flat can be dialled directly.
5. But the two numbering plans are not interchangeable. A phone that can only dial old numbers cannot reach a new number at all, and there is no prefix that converts one to the other.
6. So for a long transition every building keeps both systems, and every phone tries the new number first, falling back to the old one.

Where this comparison breaks:

7. Direct dialling is not the same as being open to callers. Removing NAT does not remove the front door. IPv6 home routers still block unsolicited incoming connections by default; the address is reachable, the firewall still decides.
8. And phone numbers are handed out by an office. IPv6 addresses are very often built by the device itself from a prefix the router announces, with no central office involved at all.

34.11.3 PLAIN – a worked example

1. An IPv6 address is 128 bits, written as eight groups of four hexadecimal digits separated by colons.
2. There are exactly two compression rules, and they are always applied in this order.
3. Rule one: in each group, drop leading zeros. Never drop trailing zeros.
4. Rule two: replace one run of consecutive all-zero groups with `::`. You may do this only once in an address, because otherwise it would be ambiguous.

```
full     2001:0db8:0000:0000:0008:0800:200c:417a
rule 1   2001:db8:0:0:8:800:200c:417a
rule 2   2001:db8::8:800:200c:417a
```

5. A second example, the loopback address, which is what `127.0.0.1` is in the new world.

```
full     0000:0000:0000:0000:0000:0000:0000:0001
rule 1   0:0:0:0:0:0:0:1
rule 2   ::1
```

6. A third, the unspecified address, which is what `0.0.0.0` is in the new world. All 128 bits are zero, so the whole thing compresses to `::`.
7. A fourth, showing the "only once" rule. This address has two separate zero runs.

```
full     2001:0db8:0000:0000:0001:0000:0000:0001
right    2001:db8::1:0:0:1
also ok  2001:db8:0:0:1::1
wrong    2001:db8::1::1        (ambiguous, illegal)
```

8. The preferred form, from RFC 5952, says compress the longest run, use lowercase letters, and do not use `::` for a single zero group. So `2001:db8::1:0:0:1` is the correct spelling of that one.
9. In a web address you must put the address in square brackets, because the colon already means "port".

```
https://[2606:4700:4700::1111]/          port 443
https://[2606:4700:4700::1111]:8443/     port 8443
```

10. Prefix lengths work exactly as in IPv4. `/64` is the normal size of one local network, and it contains 18.4 quintillion addresses. Your home is usually delegated a `/56`, which is 256 such networks.

34.11.4 PLAIN - what is really happening inside

1. When an IPv6 machine joins a network, it does not wait to be given an address. It makes one, checks it, and starts using it.
2. First it builds a link-local address beginning `fe80::`. Every IPv6 interface always has one, and it works even with no router present.
3. Then it checks nobody else has that address, by asking the network directly. This is duplicate address detection.
4. Then it sends a router solicitation, a short message meaning "is there a router here, and what prefix do we use?".
5. The router replies with a router advertisement carrying the prefix, for example `2401:4900:1f3f:abcd::/64`, and some flags.
6. The machine appends its own 64-bit interface part to that prefix and now has a global address. This is SLAAC, stateless address autoconfiguration, and no server was involved.
7. Because that interface part used to be derived from the hardware address, it followed you between networks and could track you. Modern systems therefore generate random, periodically changing addresses instead.
8. A machine normally ends up with several addresses at once: one link-local, one stable global, one or more temporary globals, and possibly one from DHCPv6. This is normal and not a fault.
9. ARP does not exist in IPv6. Its job, turning an address into a hardware address on the local wire, is done by neighbour discovery, which uses ICMPv6 messages sent to a multicast group rather than a broadcast.
10. That is a real improvement. An ARP broadcast interrupts every device on the network. A neighbour solicitation is sent to a solicited-node multicast group that typically only the target listens to.
11. A dual-stack machine looking up a name asks for both an A record, which is IPv4, and an AAAA record, which is IPv6, at the same time.
12. If both come back, it starts connecting over IPv6, and if that has not succeeded within a short delay it starts an IPv4 attempt in parallel and uses whichever finishes first. That is Happy Eyeballs.
13. The user never sees which family won. That is the entire point of the design, and it is why broken IPv6 in 2012 caused visible page hangs and broken IPv6 today usually does not.

34.11.5 TECHNICAL - the engineer's version

1. IPv6 was first specified in RFC 1883, December 1995, by Steve Deering and Bob Hinden, revised as RFC 2460 in December 1998, and is now RFC 8200, July 2017, an Internet Standard, STD 86.
2. Address architecture is RFC 4291, February 2006. Canonical text representation is RFC 5952, August 2010. The documentation prefix `2001:db8::/32` is RFC 3849 and is the only prefix you should use in examples.
3. Address types you must recognize on sight.

Type	Prefix	IPv4 counterpart
Global unicast	2000::/3	public address
Link-local	fe80::/10	169.254.0.0/16
Unique local	fc00::/7	10.0.0.0/8
Multicast	ff00::/8	224.0.0.0/4
Loopback	::1/128	127.0.0.1
Unspecified	::/128	0.0.0.0

4. There is no broadcast address in IPv6 at all. Everything broadcast used to do is done with multicast groups such as `ff02::1`, all nodes on the link, and `ff02::2`, all routers on the link.
5. Unique local addresses are RFC 4193, October 2005. In practice only `fd00::/8` is used, with a randomly generated 40-bit global identifier, so that two merged networks do not collide. They are not routable on the public internet.
6. Link-local addresses need a zone index when you use them, because `fe80::1` is ambiguous across interfaces. Write `fe80::1%en0` on macOS or `fe80::1%eth0` on Linux.
7. SLAAC is RFC 4862. Router advertisement is ICMPv6 type 134, router solicitation is type 133. The `A` flag on a prefix option permits autoconfiguration; the `M` flag says use DHCPv6 for addresses; the `O` flag says use DHCPv6 only for other information such as DNS servers.
8. Interface identifiers were originally modified EUI-64, built by splitting the 48-bit MAC address, inserting `ff:fe` in the middle and flipping the seventh bit. Because that leaks the hardware address, RFC 8981, February 2021, which replaced RFC 4941, defines temporary addresses, and RFC 7217 defines stable opaque identifiers that differ per network.
9. DHCPv6 is RFC 8415, November 2018. It provides stateful assignment and, more importantly for home networks, prefix delegation, by which a router requests a `/56` or `/48` for the networks behind it. A long-standing practical problem is that Android has never implemented DHCPv6 address assignment, so any network that offers only DHCPv6 leaves Android phones without IPv6.
10. Neighbour discovery is RFC 4861, September 2007. Neighbour solicitation is ICMPv6 type 135, advertisement is 136. The solicited-node multicast address is `ff02::1:ff` followed by the low 24 bits of the target address. It also provides duplicate address detection and neighbour unreachability detection, which ARP never had.
11. Because so much of IPv6 depends on ICMPv6, blanket-blocking ICMP breaks IPv6 completely. RFC 4890 lists which types must be permitted. This is a common self-inflicted outage.
12. Happy Eyeballs is RFC 6555, April 2012, by Dan Wing and Andrew Yourtchenko, replaced by RFC 8305, December 2017, by David Schinazi and Tommy Pauly. The recommended resolution delay is 50 milliseconds and the recommended connection attempt delay is 250 milliseconds, with a 2 second minimum fallback. A version 3 is being written in the IETF HAPPY working group; revision 04 is dated 2 July 2026 and it is not an RFC yet.
13. Transition and exhaustion dates, which explain the pace better than any argument.

Event	Date
IANA free pool exhausted	3 February 2011
APNIC exhausted, Asia	15 April 2011
World IPv6 Launch	6 June 2012
RIPE NCC exhausted	14 September 2012
ARIN exhausted	24 September 2015
Google IPv6 passes 50%	28 March 2026

14. The honest reasons adoption took thirty years, in order of weight. IPv6 is not backward compatible, so the first mover pays and gains nothing. NAT and CIDR, RFC 1519 in 1993, relieved the pressure that was supposed to force the move. Carrier-grade NAT let ISPs grow without addresses. Every firewall, monitoring tool, log parser and internal script had to be redone. And a resale market in IPv4 addresses, at roughly 30 to 50 US dollars each in recent years, made staying put affordable.
15. Where it did happen, it happened fast and for a reason: mobile networks built after 2012 had no address supply, so they went IPv6-first with translation at the edge. Reliance Jio in India, launched in 2016, is the largest example, which is why India's IPv6 capability is around 72 percent, near the top of the world table, while its fixed broadband lags.
16. Now the reader's own line, `IPv6: (none)`. What it means precisely: the machine had no usable global IPv6 address and therefore no IPv6 default route. Almost certainly the router advertised no IPv6 prefix,

which is normal on IPv4-only home broadband behind carrier-grade NAT. The trace's private hops `172.31.0.17`, `172.26.22.235` and `172.16.18.33` are consistent with exactly that kind of network.

17. What it did not cause: the failure. `github.com` publishes no AAAA record. Checked again on 13 August 2026, `dig AAAA github.com` returns nothing while `dig A github.com` returns an address. So even a perfect dual-stack machine would have used IPv4 for that request.
18. What it did remove: a second chance. On a dual-stack host talking to a dual-stack site, Happy Eyeballs races both families, and a silent drop on one is often invisible because the other completes. With no IPv6 there was exactly one path, and when that path went silent there was nothing to fall back to.
19. It also simplified the diagnosis, which is a genuine benefit. There was one address, one path, one failure, and `curl -v` naming `20.207.73.82` was the complete story.
20. Observation commands: `ip -6 addr` and `ip -6 route` on Linux, `ifconfig` and `netstat -rn -f inet6` on macOS, `ip -6 neigh` or `ndp -an` for the neighbour cache, `ping6 -c3 ff02::1%en0` to see every IPv6 host on the link, `dig AAAA example.com` for records, and `curl -6` or `curl -4` to force a family.

34.11.6 WORDS - remember these

1. IPv6 — the long-address internet — 128-bit addressing defined by RFC 8200, not backward compatible with IPv4.
2. Dual stack — both at once — a host running IPv4 and IPv6 simultaneously and selecting per connection.
3. Link-local — an address that works with no router — `fe80::/10`, valid only on one link, requiring a zone index.
4. Unique local — private addressing for IPv6 — `fc00::/7`, in practice `fd00::/8` with a random global identifier, RFC 4193.
5. SLAAC — the device builds its own address — stateless autoconfiguration from a router-advertised prefix, RFC 4862.
6. Router advertisement — the router announcing the prefix — ICMPv6 type 134, carrying prefix, flags and lifetimes.
7. Neighbour discovery — IPv6's replacement for ARP — ICMPv6 types 133 to 137, using solicited-node multicast instead of broadcast.
8. AAAA record — the IPv6 version of an A record — a DNS record mapping a name to a 128-bit address.
9. Happy Eyeballs — try both families and take the winner — the connection racing algorithm of RFC 8305.
10. Prefix delegation — your router being given a block — DHCPv6-PD handing a site a `/56` or `/48` to subnet internally.

34.12 Multicast, broadcast and mDNS

34.12.1 PLAIN - in simple words

1. There are three ways to address a message on a network, and every beginner meets all three without being told they are different.
2. **Unicast** is one to one. You address one machine, and only that machine receives it. This is almost all traffic.
3. **Broadcast** is one to everybody on this local network. Every device receives it and must look at it, even if it does not care.
4. **Multicast** is one to a named group. Only devices that joined the group receive it. Everyone else ignores it, ideally without even being interrupted.
5. Broadcast never crosses a router. It stops at the edge of your local network, and that is by design, or the internet would drown.
6. Multicast can in principle cross routers, and inside company and provider networks it does. Across the public internet it essentially never does.

7. Broadcast is how your machine gets an address when it joins a network, and how it finds the hardware address of its neighbours.
8. Multicast is how devices announce what they are: printers, speakers, televisions, game consoles, other laptops.
9. **mDNS** is ordinary DNS, done over multicast, with no server anywhere. It is why names ending in `.local` work, and why your phone finds a printer you never configured.

34.12.2 PLAIN - a picture in your head

1. Picture a classroom full of people.
2. Unicast is walking over and speaking to one person quietly.
3. Broadcast is standing at the front and shouting to the whole room. Everybody must stop and listen long enough to decide whether it concerns them.
4. Multicast is saying "this is for the chess club only". You still say it out loud, but only the chess club is expected to pay attention.
5. mDNS is somebody shouting "does anyone here have a printer?", a printer shouting back "I do, and here is where to find me", and everyone else in the room writing that down for later because they heard it anyway.
6. That last detail is real and clever. Answers are shouted, not whispered, so every device updates its notes from one exchange.

Where this comparison breaks:

7. A shout does not leave the room, and neither does broadcast. But many cheap network switches treat multicast exactly like a shout and send it to every port anyway, which defeats the point. Better switches watch who joined which group and only forward accordingly.
8. And a classroom has one room. A home with a guest network, or an office with several VLANs, is several rooms with a closed door between them, which is why discovery mysteriously stops working the moment somebody adds a second network.

34.12.3 PLAIN - a worked example

1. Finding a printer on a home network, with no configuration at all.

```
1 laptop asks    PTR  _ipp._tcp.local
2 printer says   PTR  Office Printer._ipp._tcp.local
3 laptop asks    SRV and TXT for that name
4 printer says   SRV  port 631, host prn9010.local
5 printer says   TXT  rp=ipp/print, pdl=image/urf
6 laptop asks    A    for prn9010.local
7 printer says   A    192.168.0.31
8 laptop prints  to 192.168.0.31 port 631
```

2. Every one of those questions and answers went to the multicast group `224.0.0.251` on UDP port 5353. No DNS server was involved at any point.
3. You can watch all of it happen. On macOS the tool is built in.

```
dns-sd -B _services._dns-sd._udp local.   # what kinds exist
dns-sd -B _ipp._tcp                       # printers
dns-sd -B _airplay._tcp                   # AirPlay devices
dns-sd -L "Office Printer" _ipp._tcp      # full details
ping prn9010.local                        # name to address
```

4. On Linux the equivalent comes from Avahi.

```
avahi-browse -art
avahi-resolve -n prn9010.local
```

5. Service names follow a fixed shape: an instance name, an underscore service type, an underscore protocol, then `local`.

Service type	What it is
`_ipp._tcp`	printing
`_airplay._tcp`	AirPlay video
`_raop._tcp`	AirPlay audio
`_googlecast._tcp`	Chromecast
`_ssh._tcp`	SSH servers
`_hap._tcp`	HomeKit accessories

6. So "it just works" is not magic. It is four record types, one multicast group and one well-known port.

34.12.4 PLAIN - what is really happening inside

1. Broadcast at the hardware level means sending to the Ethernet address `ff:ff:ff:ff:ff:ff`. Every network card accepts a frame with that destination and passes it up.
2. That is why ARP and the first steps of DHCP use broadcast: at that moment the sender does not know who to ask, or does not have an address yet.
3. Multicast at the hardware level uses a different set of Ethernet addresses, derived from the group address. A card is told which of them to accept, and it silently discards the rest without waking the processor.
4. That is the real saving. Broadcast costs every device time. Multicast costs only the members, provided the switch and the network card do their part.
5. Devices announce group membership so switches and routers know where to send. In IPv4 this is IGMP; in IPv6 it is MLD. Switches that listen in on those messages are said to do snooping, and that is what stops multicast becoming broadcast.
6. mDNS uses this. Every participating device joins one group, listens on one port, and answers questions about itself only.
7. Answers are sent to the group, not to the asker, so every listener can cache them. A busy network therefore needs far fewer queries than you would expect.
8. Names are claimed, not assigned. A device probes its desired name three times, and if someone else answers, it renames itself, typically by adding a number. That is where `MacBook-2.local` comes from.
9. Records carry a flag meaning "replace what you have", so corrections propagate. A device leaving politely sends its records with a lifetime of zero, which is a goodbye.
10. The packets are deliberately built so they do not travel. Their time-to-live is set so a router will not forward them, and the group is a link-local group. mDNS is not designed to cross networks, and does not.
11. That is the single biggest source of "it worked at home and not at the office". Two subnets, or a guest network with client isolation turned on, and discovery stops instantly with no error message.
12. A full-tunnel VPN can do the same thing, by capturing name lookups or the default route. On the reader's macOS machine, `utun` interfaces were present, and that class of interface is exactly where such interference would appear.

34.12.5 TECHNICAL - the engineer's version

1. The addresses worth memorizing.

Address	Meaning
255.255.255.255	limited broadcast
224.0.0.1	all hosts on this link

Address	Meaning
224.0.0.2	all routers on this link
224.0.0.251	mDNS over IPv4
ff02::fb	mDNS over IPv6
239.255.255.250	SSDP and WS-Discovery

2. IPv4 multicast is `224.0.0.0/4`. Within it, `224.0.0.0/24` is link-local control traffic that routers never forward, and `239.0.0.0/8` is administratively scoped for private use, defined in RFC 2365, July 1998.
3. IPv6 has no broadcast at all. `ff02::1` is all nodes on the link and `ff02::2` is all routers, and scope is encoded in the address itself.
4. Ethernet mapping: an IPv4 group maps to a MAC address of `01:00:5e` plus the low 23 bits of the group. Because 32 different groups share one MAC address, a card can accept traffic it did not want, and software must filter again.
5. Group membership signalling: IGMPv1 is RFC 1112, August 1989, IGMPv2 is RFC 2236, November 1997, IGMPv3 is RFC 3376, October 2002, which added source-specific multicast. For IPv6 the equivalent is MLDv2, RFC 3810, June
2004. IGMP snooping guidance is RFC 4541, May 2006, and is informational, which is why switch behaviour varies so much.
6. Discovery protocols on a normal home network.

Protocol	Port	Group
mDNS	UDP 5353	224.0.0.251
LLMNR	UDP 5355	224.0.0.252
SSDP	UDP 1900	239.255.255.250
WS-Discovery	UDP 3702	239.255.255.250

7. mDNS is RFC 6762 and DNS Service Discovery is RFC 6763, both February 2013, both by Stuart Cheshire and Marc Krochmal of Apple. Apple shipped the implementation as Rendezvous in Mac OS X 10.2 in August 2002 and renamed it Bonjour in 2005 after a trademark dispute. Avahi is the Linux equivalent, first released in 2005. Windows has shipped a native mDNS responder since Windows 10.
8. `.local` is reserved by RFC 6762 for mDNS. Using `.local` as an internal Active Directory domain name, which was once common advice, produces a permanent low-level conflict on any network with Apple devices.
9. Service enumeration uses the meta-query `_services._dns-sd._udp.local`, which returns a PTR record per service type present. Then PTR gives instances, SRV gives host and port, TXT gives key-value parameters, and A or AAAA gives the address. Four record types, in that order, every time.
10. SSDP is part of UPnP, which Microsoft introduced in 1999. It uses HTTP-like text over UDP with `M-SEARCH` and `NOTIFY` verbs. It has been widely abused for reflection and amplification attacks, which is why UDP 1900 should never be reachable from the internet.
11. WS-Discovery on UDP 3702 is what modern Windows and modern network printers use after Microsoft deprecated the old NetBIOS browsing and SMBv1.
12. The Wi-Fi cost, which surprises people. Multicast and broadcast frames are normally sent at the lowest basic rate so every client can hear them, and they are not acknowledged. On a busy access point, chatty discovery traffic therefore consumes disproportionate airtime. Enterprise access points respond by converting multicast to unicast, proxying mDNS, or dropping it, and each of those changes behaviour in a different way.
13. Multicast routing across the public internet was specified, deployed experimentally as the MBone from 1992, and never became general service. PIM Sparse Mode is RFC 7761, March 2016, and is used inside

provider networks for IPTV and inside exchanges for market data, but you cannot assume any multicast reaches the internet.

14. Observation: `tcpdump -ni en0 udp port 5353` shows the traffic, `dns-sd -B` browses, `avahi-browse -art` dumps everything, `netstat -gn` or `ip maddr show` lists joined groups, and `ping -c3 224.0.0.1` reveals which hosts on the link answer at all.

34.12.6 WORDS - remember these

1. Unicast — one to one — a packet addressed to a single interface.
2. Broadcast — one to all on this link — IPv4 only, `255.255.255.255` or the subnet's all-ones address, never forwarded by routers.
3. Multicast — one to a group — delivery to hosts that joined a group address in `224.0.0.0/4` or `ff00::/8`.
4. IGMP — how a host joins a group — Internet Group Management Protocol, RFC 3376 for version 3; MLD is the IPv6 equivalent.
5. Snooping — the switch learning who joined — IGMP or MLD snooping, so multicast is not flooded to every port.
6. mDNS — DNS with no server — RFC 6762 multicast DNS on UDP 5353 serving the `.local` namespace.
7. DNS-SD — how services are named and found — RFC 6763 conventions using PTR, SRV and TXT records over unicast or multicast DNS.
8. Bonjour — Apple's name for it — the Apple implementation of mDNS and DNS-SD, shipped as Rendezvous in 2002.
9. SSDP — the older discovery protocol — UPnP's search mechanism on UDP 1900, a frequent amplification-attack vector.

34.13 Network performance for developers

34.13.1 PLAIN - in simple words

1. When a page feels slow, "the network is slow" is not a diagnosis. It is a place to start looking.
2. Every web request is made of five separate waits, and they have completely different causes and fixes.
3. **DNS**: turning the name into an address. Slow if the resolver is far away or the record is not cached.
4. **Connect**: the handshake that opens the connection. Costs one round trip, always, no exceptions.
5. **TLS**: agreeing on encryption. Costs one more round trip on modern versions, two on older ones.
6. **Time to first byte**: you asked, and you are waiting for the server to think. This includes one more round trip plus the server's own work.
7. **Transfer**: the bytes arriving. This is the only part that bandwidth actually improves.
8. Most slow requests are the first four. Most "optimizations" people try improve only the fifth.
9. The good news is that you can measure all five with one command, on any machine, in one line, and stop guessing forever.

34.13.2 PLAIN - a picture in your head

1. Think of telephoning a shop in another city to ask whether they have an item.
2. Looking up the number is DNS.
3. Dialling and waiting for someone to pick up is the connection handshake.
4. Agreeing that you will both speak in a private code, and checking the other side really is that shop, is TLS.
5. Asking your question and waiting while they walk to the back room is time to first byte.
6. Them reading the answer aloud is the transfer.
7. If the shop is far away, every single exchange in that conversation takes longer, because every sentence travels the distance twice.
8. So the way to make the call quick is to ask fewer questions, not to speak faster.

Where this comparison breaks:

9. A phone call is one conversation. Loading one page opens dozens, and older protocols could only ask one question per line at a time, which is exactly why browsers used to open six lines to the same shop.
10. And you can keep a phone line open for the next question. Programs often fail to, hanging up and redialling for every item, which is the single most common self-inflicted performance bug.

34.13.3 PLAIN – a worked example

1. `curl` can print the exact time each phase finished. Put the format in a file, because it is easier to read.

```
cat > fmt.txt <<'EOF'
   dns  %{time_namelookup}s
  conn  %{time_connect}s
   tls  %{time_appconnect}s
  send  %{time_pretransfer}s
  ttfb  %{time_starttransfer}s
 total  %{time_total}s  code %{http_code}
EOF
curl -sS -o /dev/null -w @fmt.txt https://github.com/
```

2. A healthy result from an Indian connection to a nearby edge looks like this.

```
   dns  0.028s
  conn  0.081s
   tls  0.190s
  send  0.191s
  ttfb  0.352s
 total  0.404s  code 200
```

3. Those numbers are cumulative, all measured from the start. The useful quantities are the differences.

Phase	Arithmetic	Time
DNS	namelookup	28 ms
TCP handshake	conn minus dns	53 ms
TLS handshake	tls minus conn	109 ms
Server thinking	ttfb minus send	161 ms
Download	total minus ttfb	52 ms

4. Read it. The round trip to this server is about 53 milliseconds, because that is what a TCP handshake costs. TLS cost roughly two of those. Actual data transfer was 52 milliseconds out of 404.
5. So 87 percent of that request was setup and waiting. Buying a faster line improves the 52 milliseconds and nothing else.
6. Now the same command during the reader's own outage. The shape of the failure is the diagnosis.

```
   dns  0.026s
  conn  0.000s
   tls  0.000s
  ttfb  0.000s
 total  15.001s  code 000
```

7. DNS succeeded, so the name resolved to `20.207.73.82` correctly. Then every later phase is zero, because none of them was ever reached.
8. A zero in a later phase does not mean it was fast. It means it never happened. `code 000` means no HTTP response existed at all.
9. That is the signature of a silent drop: the connection phase never completed and nothing came back, not a refusal, not an error. Compare it with a refused connection, which fails in a few milliseconds with

`code 000` and a different curl exit code, and with a slow server, which shows a large `ttfb` and a normal `code 200`.

10. Three different faults, three different shapes, one command.

34.13.4 PLAIN - what is really happening inside

1. The first fix is to stop paying the setup cost repeatedly. If you close the connection after each request, the next request pays DNS, TCP and TLS again.
2. Keeping the connection open is called keep-alive, and it has been the default in HTTP since version 1.1. But a program that creates a new client object per request defeats it, and this is very common in application code.
3. The second fix is to stop queuing. In HTTP/1.1, one connection carries one request at a time. Browsers worked around it by opening about six connections per host, which multiplies handshakes and confuses congestion control.
4. HTTP/2 solves this properly by carrying many requests inside one connection as interleaved streams. One handshake, one congestion window, dozens of requests in flight.
5. Header compression helps too. The same cookies and headers repeat on every request; HTTP/2 sends them once and refers back to them afterwards.
6. But HTTP/2 has one honest weakness. All those streams still ride one TCP connection, so a single lost packet stalls every stream until it is resent. That is head-of-line blocking, moved down a layer rather than removed.
7. HTTP/3 fixes that by running over QUIC on UDP, where each stream is recovered independently, and a loss affecting one stream does not stall the others.
8. The third fix is compression. Text compresses enormously, so a compressed response transfers in a fraction of the time. It costs a little processor time on both ends and nothing else.
9. Compression only shortens the transfer phase. It does not remove a single round trip, which is why it helps large responses and barely touches small ones.
10. The fourth fix is doing setup earlier. A browser can be told to resolve a name, or open a connection, before it knows it will need it, so the cost is paid during time the user was going to spend anyway.
11. The last and largest fix is asking for less. A request you do not make costs nothing and cannot fail.

34.13.5 TECHNICAL - the engineer's version

1. The `curl` write-out variables, all in seconds from the start of the transfer.

Variable	Marks the moment
`time_namelookup`	name resolution done
`time_connect`	TCP handshake done
`time_appconnect`	TLS handshake done
`time_pretransfer`	about to send request
`time_starttransfer`	first response byte
`time_total`	transfer complete

2. Useful companions: `%{http_version}`, `%{num_connects}`, `%{num_redirects}`, `%{time_redirect}`, `%{size_download}`, `%{speed_download}` and `%{remote_ip}`. `curl -w` also accepts `%{json}` in version 7.70.0 and later, which prints every variable as one JSON object.
3. A one-line version worth keeping in your shell history: `curl -so /dev/null -w 'dns %{time_namelookup} tls %{time_appconnect} ttfb %{time_starttransfer} tot %{time_total}\n' https://example.com/`.
4. Connection reuse. HTTP/1.1, RFC 2068 in January 1997, made persistent connections the default. In practice the wins come from client configuration: reuse one HTTP client object, set a connection pool size, and keep idle connections alive for longer than your request interval.

5. HTTP/2 is RFC 7540, May 2015, restated as RFC 9113, June 2022. It requires ALPN negotiation during the TLS handshake, uses HPACK header compression, RFC 7541, and multiplexes streams over one connection. Server push was part of it, was found not to help, and Chrome removed support in version 106 in October 2022.
6. HTTP/3 is RFC 9114, June 2022, over QUIC, RFC 9000, May 2021, with QPACK header compression, RFC 9204. It removes transport-level head-of-line blocking, survives a change of network address through connection identifiers, and includes the TLS 1.3 handshake in its own setup.
7. Compression, with real behaviour. gzip is RFC 1952 and is universal. Brotli is RFC 7932, July 2016, and typically produces files 15 to 20 percent smaller than gzip on JavaScript and CSS at its highest setting, at a much higher compression cost, so it suits static assets compressed once. Zstandard is RFC 8878, February 2021, and its value is speed at a similar ratio, which suits dynamic responses; browser support arrived in Chrome 123 in March 2024 and Firefox 126 in May 2024.
8. Never compress content that is already compressed, such as images, video or archives. You spend processor time to add a few bytes.
9. TLS costs, precisely. TLS 1.2, RFC 5246, needs two round trips. TLS 1.3, RFC 8446, August 2018, needs one, and a resumed session can send data in the first flight, which is 0-RTT, at the price of replay exposure for non-idempotent requests. Session resumption tickets are what make the second visit cheap.
10. Browser hints that buy round trips back, all standard HTML today: `dns-prefetch` resolves a name early, `preconnect` completes DNS, TCP and TLS early, `preload` fetches a known-critical asset early, and `Early Hints`, status code 103 from RFC 8297, December 2017, lets the server send those hints before the real response is ready.
11. Server-side measurement should be exported, not guessed. The `Server-Timing` response header, a W3C specification, carries named durations from the server into the browser's own performance timeline.
12. Rough real-world weights for a typical page on a 200 millisecond path, approximate and workload-dependent.

Change	Typical effect
Connection reuse	removes 2 RTT per request
HTTP/1.1 to HTTP/2	removes most queueing
gzip or brotli on text	60 to 80 percent smaller
CDN edge for static	removes most of the RTT
One fewer redirect	removes a full request

13. Measurement discipline. Test from where the users are, not from the datacentre. Test with a cold cache and a warm cache separately. Test the 99th percentile, because the mean hides the failures. And prefer real user monitoring over one laptop's opinion.

34.13.6 WORDS - remember these

1. Time to first byte — how long before anything arrives — the interval from request sent to first response byte, including one round trip.
2. Keep-alive — do not hang up — HTTP persistent connections, default since HTTP/1.1.
3. Multiplexing — many requests, one connection — interleaved streams within a single HTTP/2 or HTTP/3 connection.
4. Head-of-line blocking — one stuck item stalls the rest — a lost segment delaying all streams sharing a TCP connection.
5. ALPN — deciding which HTTP version during the handshake — Application-Layer Protocol Negotiation, a TLS extension, RFC 7301.
6. Brotli — a stronger text compressor — RFC 7932 encoding, advertised as `br` in `Accept-Encoding`.

7. Preconnect — pay the setup before you need it — a resource hint that performs DNS, TCP and TLS for a host in advance.
8. 0-RTT — sending data in the first packet — TLS 1.3 early data on a resumed session, with replay risk.

34.14 Cloud networking preview

34.14.1 PLAIN - in simple words

1. Everything in this chapter also exists inside a cloud account, with different names and a control panel instead of a cable.
2. A **VPC**, a virtual private cloud, is your own private network inside the provider's network. Nobody else's machines are in it.
3. A **subnet** is a slice of that network, usually tied to one physical zone.
4. A **route table** decides where traffic from a subnet goes. A subnet is called public or private purely because of what its route table says.
5. A **security group** is a firewall attached to a machine, and it remembers connections it allowed.
6. A **network ACL** is a firewall attached to a subnet, and it remembers nothing, so you must write both directions yourself.
7. A **load balancer** is a managed reverse proxy: one address in front of many machines.
8. **Private link** lets you reach a service without your traffic ever touching the public internet.
9. Chapter 43 builds all of this properly. This section is only so the names are not new when you get there.

34.14.2 PLAIN - a picture in your head

1. A VPC is a floor of an office building that you have rented.
2. Subnets are the rooms on that floor, each in a different fire zone.
3. The route table is the sign at each room door saying which corridor to use to leave.
4. A security group is a lock on each desk. A network ACL is a lock on the room door.

Where this comparison breaks:

5. A door lock does not care whether you are leaving or entering as a reply to something. A security group does, and a network ACL does not, and that single difference causes most cloud firewall confusion.

34.14.3 PLAIN - a worked example

1. A minimal two-tier layout, written as a plan.

```
VPC            10.0.0.0/16
 public subnet 10.0.1.0/24  -> route 0.0.0.0/0 to IGW
 private subnet 10.0.2.0/24 -> route 0.0.0.0/0 to NAT
 load balancer in public subnet, port 443
 app servers in private subnet, port 8080
 database in private subnet, port 5432
```

2. The security groups, written as sentences rather than tables.

```
sg-lb   allow in  443 from 0.0.0.0/0
sg-app  allow in  8080 from sg-lb
sg-db   allow in  5432 from sg-app
```

3. Note what is missing. No outbound rules were needed, because security groups are stateful, so the reply to an allowed request is allowed automatically.
4. Note also that rules refer to other groups, not to addresses. That is the feature worth learning first, because it survives machines being replaced.

34.14.4 PLAIN - what is really happening inside

1. None of this is a physical network. It is a software overlay running on the provider's real network, and the encapsulation is exactly the tunnelling from section 34.8.
2. Your packets are wrapped, carried across shared hardware, and unwrapped, and the wrapping is what keeps your VPC separate from everyone else's.
3. A security group is enforced at the virtual network interface of the machine itself, before the packet reaches the guest, which is why the machine's own firewall never sees a blocked packet.
4. A network ACL is enforced at the subnet boundary, so traffic between two machines in the same subnet never meets it.
5. Because security groups only allow and never deny, they cannot express "allow everyone except this address". Network ACLs can, which is the main reason they still exist.
6. A private link endpoint puts an address from your own subnet in front of somebody else's service, so the traffic stays inside the provider's network and never needs an internet gateway at all.

34.14.5 TECHNICAL - the engineer's version

1. Amazon launched Virtual Private Cloud in August 2009, and made a default VPC automatic for new accounts in 2013. Azure calls it a virtual network, and Google calls it a VPC network, which is global rather than regional.
2. The comparison that matters most in interviews and in incidents.

Property	Security group	Network ACL
Attached to	an interface	a subnet
Stateful	yes	no
Rule types	allow only	allow and deny
Evaluation	all rules	numbered, first match

3. AWS reserves five addresses in every subnet: the network address, the VPC router, the DNS address at the second host address, one reserved for future use, and the broadcast address. So a `/24` gives 251 usable addresses, not 254. VPC CIDR blocks may be `/16` to `/28`.
4. Load balancer types, with dates. The Classic Load Balancer arrived in 2009, the Application Load Balancer, which is layer 7 and understands HTTP, in August 2016, the Network Load Balancer, layer 4, in September 2017, and the Gateway Load Balancer, which uses Geneve encapsulation, in November 2020.
5. AWS PrivateLink was announced in November 2017. Gateway endpoints for object storage predate it. The distinction to remember is that an interface endpoint consumes an address in your subnet and a gateway endpoint is a route table entry.
6. Connectivity between networks: peering is non-transitive, so three peered VPCs do not form a mesh; Transit Gateway, announced November 2018, exists precisely to solve that.
7. Egress costs money and ingress usually does not, which is a commercial fact with architectural consequences, and it is why cross-zone chatter shows up on bills.
8. Observation: `aws ec2 describe-security-groups`, `aws ec2 describe-route-tables`, and VPC Flow Logs, which record accepted and rejected flows and are the cloud equivalent of the packet capture you would otherwise take.
9. Chapter 43 covers the whole of this properly, including NAT gateways, IPv6- only subnets, and how a request reaches a container.

34.14.6 WORDS - remember these

1. VPC — your own private network in the cloud — a logically isolated virtual network with your own address range.
2. Subnet — a slice of that network in one zone — a CIDR range bound to an availability zone with its own route table association.
3. Route table — the sign saying which way out — a set of destination-to-target rules applied to a subnet.
4. Security group — a stateful firewall on the machine — allow-only rules evaluated at the elastic network interface.
5. Network ACL — a stateless firewall on the subnet — numbered allow and deny rules evaluated in order, in both directions.
6. Load balancer — one public address, many servers — a managed layer 4 or layer 7 reverse proxy with health checks.
7. Private link — reach a service without the internet — an endpoint placing a provider service on an address inside your own subnet.

34.98 Common wrong ideas

1. Wrong: `127.0.0.1` and `0.0.0.0` are two ways of saying the same thing. Right: `127.0.0.1` means this machine only, and listening on `0.0.0.0` means every interface the machine has, which is the opposite of private.
2. Wrong: a firewall makes a machine secure. Right: a firewall only decides which packets are allowed to arrive. The service listening behind it must still be safe on its own, because allowed traffic is not checked.
3. Wrong: DROP and REJECT are the same, since both block the packet. Right: REJECT sends a refusal and fails in milliseconds, DROP says nothing and fails at your timeout. The reader's 15 seconds of silence was the DROP signature.
4. Wrong: `* * *` at the end of a traceroute proves the traffic is blocked there. Right: many routers deprioritize or never send ICMP time-exceeded replies, so silence at the end of a trace is normal and proves nothing.
5. Wrong: a VPN makes you anonymous. Right: it moves the point where your traffic becomes visible, from your ISP to your VPN provider, and it changes nothing about cookies, logins or browser fingerprints.
6. Wrong: a proxy handling your HTTPS traffic can read it. Right: after a CONNECT it copies bytes blindly and sees the hostname, sizes and timing only, unless it has been made a trusted certificate authority on your machine.
7. Wrong: a CDN makes everything faster. Right: it makes cacheable things faster and a first miss slightly slower, and it does nothing at all for a response that is different for every user.
8. Wrong: anycast is a kind of DNS load balancing. Right: anycast is one address announced from many places by BGP, decided by routers. DNS steering is different addresses given to different askers, decided by a name server.
9. Wrong: a slow website is fixed by buying more bandwidth. Right: page load is usually latency-bound, so it is fixed by removing round trips, moving content closer, and reusing connections.
10. Wrong: IPv6 is IPv4 with more numbers, so machines can talk to each other across the two. Right: they are separate protocols with no interoperability, which is why dual stack and Happy Eyeballs exist.

34.99 Chapter summary in 20 lines

1. `127.0.0.1` is an address meaning this machine, `localhost` is a name for it, and `0.0.0.0` is a placeholder that means all interfaces when listening and unknown when sending.
2. A service bound to loopback is invisible from every other machine, and that is the address behaving correctly, not a firewall.

3. Ports are 16-bit numbers, split by convention into well known below 1024, registered to 49151, and ephemeral above that, and a busy machine exhausts the ephemeral range long before it runs out of anything else.
4. A firewall decides which packets are allowed. It does not inspect what an allowed packet then does, and it is not a substitute for a safe service.
5. Stateless filtering judges each packet alone; stateful filtering remembers connections, which is why you write one rule instead of two.
6. REJECT refuses immediately and produces an instant error. DROP says nothing and produces a timeout, which is why the reader's `curl` sat silent for 15 seconds against `20.207.73.82` and then gave up.
7. The same request succeeding over mobile data proved the server was up and the fault lay on one path, while the trace ending in `* * *` after `ae106-0.rwa04.pnq20.ntwk.msn.net` proved nothing at all on its own.
8. Middleboxes sit between every client and server, and each one can see a different amount: addresses, ports, hostnames from SNI, or full content if it terminates encryption.
9. A VPN encapsulates whole IP packets at layer 3, so every program is covered, and macOS shows those tunnels as `utun` interfaces.
10. A proxy works at layer 5 to 7 for the connections that were configured to use it, so a forward proxy hides clients and a reverse proxy hides servers.
11. Tunnelling is putting one packet inside another, and it always costs overhead, which is why broken path MTU discovery produces the classic small-pages-work, large-pages-hang fault.
12. SSH gives three tunnels: `-L` opens a port on your side, `-R` opens a port on theirs, and `-D` gives a SOCKS proxy for anything.
13. Services such as ngrok and Cloudflare Tunnel expose a local port publicly by dialling outward first, which is why they work behind a home router with no port forwarding.
14. A CDN caches what is identical for everybody, uses a cache key built from scheme, host, path and query, and turns a 600 millisecond origin fetch into a 45 millisecond edge hit.
15. Anycast is one address announced from many sites by BGP; it suits DNS and UDP perfectly, works for TCP because routes are stable rather than pinned, and breaks a live connection when routing changes mid-flow.
16. DNS steering is the other mechanism entirely: different answers to different askers, limited by TTLs and by the fact that the server sees the resolver, such as the reader's `1.1.1.1`, and not the user.
17. Latency is bounded by the speed of light in fibre, about 200,000 kilometres per second, so India to United States East cannot go below about 128 milliseconds and measures 185 to 200 in practice.
18. Bandwidth-delay product is bandwidth times round-trip time, so a 100 megabit link at 200 milliseconds needs 2.5 megabytes in flight, and page load time is dominated by round trips rather than by capacity.
19. IPv6 addresses are 128 bits with two compression rules, use SLAAC and neighbour discovery instead of DHCP and ARP, and the reader's `IPv6: (none)` cost them nothing directly because `github.com` publishes no AAAA record, though it removed the second path Happy Eyeballs would have raced.
20. Broadcast reaches everyone on a link, multicast reaches a group, mDNS on `224.0.0.251` port 5353 makes `.local` discovery work with no server, and `curl -w` measures DNS, connect, TLS, first byte and transfer separately so that "the network is slow" becomes an actual number.

ABOUT THE AUTHOR

Shikhar Singh

Founder & Chairman

KedByte Technologies Private Limited

Shikhar Singh is the Founder & Chairman of KedByte Technologies Private Limited.

He wrote this book to solve a problem he kept meeting: capable people concluding they were not technical, when in truth nobody had ever shown them the bottom of the stack. His conviction is that computing has no genuinely hard idea in it, only a very long chain of simple ones, and that anybody who is walked along that chain in order can hold the whole of it in their head.

That conviction shapes the method used here. Explain it plainly, with a comparison and a worked example, so that it lands. Then explain it again in the exact professional vocabulary, with the real units and the real figures, so that it is usable. Say clearly where the plain version stops being true. Never let a reader carry away a confident misunderstanding.

Under his direction, KedByte Technologies Private Limited treats technical education as infrastructure rather than as a product feature: knowledge that should be transferable, verifiable, and free of the vocabulary barriers that keep capable people out of the field.

ABOUT THE PUBLISHER

KedByte Technologies Private Limited

KedByte Technologies Private Limited publishes technical work under the KedByte Press imprint.

The editorial standard applied to this volume is simple to state and difficult to meet. Every explanation must work twice: once for a reader who has never opened a terminal, and once for an engineer who needs the exact figure. Every claim that can be checked must be checked. Every figure that will go stale must carry a date. Where the field genuinely disagrees, both positions are given rather than one being quietly chosen.

Where this book uses real evidence, it uses it unaltered. The network trace in Part F, the failed pushes in Part G and the error messages quoted throughout are reproduced exactly as they were observed. Diagnosis is taught against ambiguous real data, because that is the only kind there is.

KEDBYTE

TECHNOLOGIES PRIVATE LIMITED

COLOPHON

This first edition of *How Computers Work* runs to nine parts and fifty-three chapters, and was set entirely from plain text.

The body text is set in TeX Gyre Pagella, a digital revival of Hermann Zapf's Palatino, chosen for long-form reading. Headings, tables and the KedByte identity are set in Poppins. Code, command output and diagrams are set in DejaVu Sans Mono, and every diagram in this book is drawn in plain ASCII so that it survives being copied anywhere.

The book was composed with pandoc and typeset with XeLaTeX on A4 stock. The single accent colour used on rules, chapter numbers and section headings is KedByte navy.

Shikhar Singh

Founder & Chairman, KedByte Technologies Private Limited

KB

First Edition • KedByte Press