

COMPANION VOLUME • STAGE 06 OF THE ENGINEER'S SYLLABUS

The Engineer's Exchange

An exchange is where lines meet and messages are routed: the telephone exchange that connected one caller to another, the internet exchange where networks hand traffic across, and the request-and-response exchange behind every page you load. Almost everything you build is reached over a network, and almost every mysterious slowness lives in it. This volume follows that message from the HTTP you write, down through TCP and IP to the wire, and back up again.

CHAPTERS

20

CODE EXAMPLES

23

EXERCISES

60

STAGE BUDGET

120 h

Contents

How to use this volume	4
Part I · The web	5
Chapter 01 · The network, top-down	6
Chapter 02 · URLs and the web model	12
Chapter 03 · HTTP: the message	17
Chapter 04 · HTTP: caching, cookies and negotiation	23
Chapter 05 · HTTP/2 and HTTP/3	28
Part II · Secure and real-time	34
Chapter 06 · TLS and HTTPS	35
Chapter 07 · Real-time transports	41
Chapter 08 · RPC and schemas	46
Part III · Transport	52
Chapter 09 · TCP: connection and reliability	53
Chapter 10 · TCP: flow and congestion control	58
Chapter 11 · UDP, and when losing packets is right	63
Chapter 12 · Sockets: the API underneath everything	68
Part IV · Addressing and naming	73
Chapter 13 · IP: addressing and subnets	74
Chapter 14 · Routing and NAT	79
Chapter 15 · DNS: the internet's directory	85
Part V · Delivery at scale	91
Chapter 16 · Latency, bandwidth and the physics	92
Chapter 17 · CDNs, edge and anycast	98
Chapter 18 · Load balancing at layers 4 and 7	103
Part VI · Operating the network	108
Chapter 19 · Diagnosing the network	109

Chapter 20 · What happens when you type a URL	115
The twenty rules	120
Reading list	122

The chapters follow the shape of the rest of the series: the idea in one sentence, worked examples, the traps, then exercises with folded solutions, a checklist and further reading. Networking rewards doing over reading, so run the examples, point the tools at real servers, and watch the packets.

We go top-down. The classic networking course teaches the wire first and the web last; this one, like Kurose and Ross, starts where your work actually touches the network, HTTP, and descends: the web, then transport (TCP and UDP), then addressing (IP), then naming (DNS), then delivery at scale, then how to debug it all. The layer you'll meet first in your career is the layer we meet first here. The one prerequisite is a little computer architecture (stage 04): what a process, a port and a buffer are.

The examples use the real network, or stand in for it deterministically. Where an example can run against your own machine, over real sockets, with `ipaddress`, with a local HTTP server, it does, and it was executed before publication. Where it would depend on a live remote host (a real DNS answer, a real traceroute), it's shown as captured output and marked illustrative, because that output changes by the hour and your network isn't ours. Everything you can reproduce, you should.

Tools and numbers age; the shape doesn't. The adoption percentages, the fastest resolver, the default congestion-control algorithm, these are late-2026 snapshots that will drift, and they're here to show you how to find today's answer, not to memorise. What lasts is the model: the layers, the round trip, the handshake, the name lookup. The stage's deliverable is the real test: write an HTTP/1.1 server directly on TCP sockets, handling keep-alive, chunked encoding and concurrent connections, with no framework, then implement a working TCP stack in the CS 144 labs. Build the thing, and the layers stop being a diagram.

THE ONE QUESTION

When something is slow or broken over a network, the instinct is to blame your code. Usually your code is fine and the network is doing something you didn't picture, because you pictured it as instant and direct, and it is neither.

So the question to ask of any networked operation, before you profile a single function, is: **how many round trips does this take, and how far does each one travel?** A round trip to a server across an ocean is 150 milliseconds no matter how fast your code is, and most mysterious latency turns out to be round trips you didn't know you were making, DNS lookups, TLS handshakes, redirects, chatty APIs, across distances light itself takes real time to cross. Learn to see the journey, and the network stops being magic.

PART I

The web

Five chapters on the layer you actually touch. We start, as the best networking course does, at the top: the layered model and how a request is wrapped for its journey, then URLs, then HTTP in depth, its caching and state, and the newer versions that make it fast.

The network, top-down

3 exercises

The internet is built in layers, each solving one problem and handing the rest to the layer below. Your code writes at the top, in HTTP; by the time your message reaches the wire it has been wrapped four times over, and on the way it is read only by the layers that need it. Understand the layers and the journey, and the network stops being a black box.

Why layers

Getting a message from a program on your laptop to a program in a data centre on another continent is an enormous problem, so the internet splits it into independent sub-problems and stacks the solutions. The **application layer** decides what to say (HTTP, the request for a page). The **transport layer** gets that message to the right *program* on the far host and, with TCP, makes sure it arrives complete and in order. The **network layer** (IP) gets a packet to the right *host*, hop by hop, across any number of networks. The **link layer** moves bits across a single stretch of wire or radio to the next device. Each layer is a contract: it offers a service to the layer above and relies on the one below, so you can reason about HTTP without thinking about voltages, and swap Wi-Fi for Ethernet without changing a line of your app.

This is the **TCP/IP model**, four layers. You'll also hear the older seven-layer **OSI model**, which is mostly useful now as vocabulary: when someone says "a layer 7 load balancer" they mean it understands HTTP, and "layer 4" means it works at TCP. We'll use the four-layer model and borrow the numbers where they're handy.

Encapsulation: envelopes inside envelopes

A message doesn't move down the stack unchanged; each layer **wraps** it in a header of its own, like putting a letter in an envelope, then that envelope in a bigger one. On the far side the process reverses, each layer reads and strips its own header and hands the rest up. Here is the wrapping, with the real header sizes:

EXAMPLE encap.py · wrapping a message for its journey

```
from dataclasses import dataclass

# Each layer wraps the one above in its own header, the way envelopes nest inside envelopes.
# These are the real header sizes for a classic Ethernet / IPv4 / TCP / HTTP stack.
LAYERS = [
    ("application", "HTTP", 0, "what: the request/response you wrote"),
    ("transport", "TCP", 20, "which program: ports, plus reliability (seq, ack)"),
    ("network", "IPv4", 20, "which host: source and destination IP addresses"),
    ("link", "Ethernet", 14, "which next hop: MAC addresses, one wire-segment"),
]
```

```

def encapsulate(payload_bytes):
    """Send side: each layer prepends its header. Returns the frame size at each step."""
    size = payload_bytes
    steps = []
    for layer, proto, header, _ in LAYERS:
        size += header
        steps.append((layer, proto, header, size))
    return steps

def overhead(payload_bytes):
    total_header = sum(h for _, _, h, _ in LAYERS)
    on_wire = payload_bytes + total_header
    return total_header, on_wire, total_header / on_wire

if __name__ == "__main__":
    print("a layer only reads its own header, then hands the rest up or down:")
    for layer, proto, header, why in LAYERS:
        print(f" {layer:12} {proto:9} +{header:2}B {why}")

    print("\nencapsulating a 1-byte payload (e.g. a single keystroke):")
    for layer, proto, header, size in encapsulate(1):
        print(f" after {layer:12} ({proto}): {size:3}B on the wire")
    h, wire, frac = overhead(1)
    print(f" -> {h}B of headers to carry 1B: {frac:.0%} overhead (why tiny packets waste the network)")

    h, wire, frac = overhead(1460) # a full TCP segment (1500 MTU - 40 bytes IP+TCP)
    print(f"a full 1460B segment: {h}B headers, {wire}B on the wire, {frac:.1%} overhead")

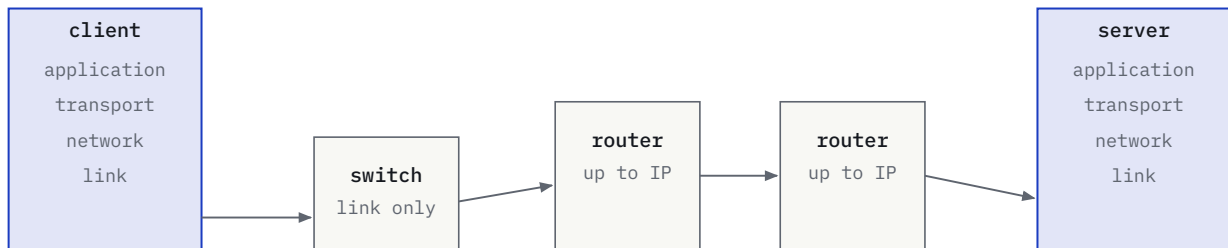
```

Two lessons fall out of the numbers. A layer reads *only its own header*: a router looks at the IP header to decide where to forward the packet and never opens the TCP or HTTP inside, which is both an efficiency and a privacy property (and why TLS, chapter 6, can hide everything above IP). And headers have a fixed cost, 54 bytes here, so carrying a single byte is 98% overhead while a full 1460-byte segment is only 3.6%: this is why chatty, tiny messages waste the network and why batching helps, a theme that returns in every later chapter.

ENCAPSULATION



THE JOURNEY



Only the two endpoints run the full stack; the network in between reads just enough to forward. This is the end-to-end principle.

A message is wrapped on the way down and unwrapped on the way up. In between, switches read only the link header and routers only up to the IP header, so the intelligence, reliability, encryption, the meaning of the request, lives at the ends, not in the middle.

The journey of a request

Follow one HTTP request. Your code hands HTTP to the transport layer, which splits it into **segments** and adds a TCP header with source and destination **ports** (so the far host knows which program the data is for) and sequence numbers (so it can be reassembled in order). Each segment goes to IP, which wraps it in a **packet** with source and destination IP **addresses** and hands it to the link layer, which frames it for the first hop. Then it travels: across your Wi-Fi to your router, which forwards it toward its destination; through your ISP; across the internet's backbone, router to router, each one reading the IP header, deciding the next hop, and passing it on; until it reaches the server's network, its host, and finally, unwrapped layer by layer, the server program. The reply makes the same trip in reverse. The whole round trip is what chapter 16 teaches you to count, because its duration is set by physics, not by your code.

The ideas that shaped it

- **Packet switching.** The internet doesn't reserve a dedicated line for your conversation the way old telephone networks did; it chops data into packets that share links with everyone else's, each routed independently. That's why the internet is cheap and resilient (a packet routes around a failure) and also why latency varies (your packet waits in queues behind others).
- **The end-to-end principle.** Keep the network in the middle simple and fast, and put the intelligence, reliability, ordering, encryption, at the endpoints. It's why IP itself makes no promises (a packet may be lost, duplicated or reordered) and TCP builds reliability on top, at the ends. This principle explains more design decisions in this volume than any other.

- **Addresses at every layer.** A thing has a different name at each layer: a human-friendly *hostname* at the application layer (resolved by DNS, chapter 15), an *IP address* at the network layer (chapter 13), a *port* at the transport layer, and a hardware *MAC address* at the link layer. Confusing which name lives where is the source of many networking muddles.

COMMON TRAPS

Imagining the network as instant and direct. It's a series of hops across real distance, each adding delay.

Mixing up the addresses: a port is not an IP, a hostname is not an IP, a MAC is not routable across the internet.

Thinking routers understand your request. They read the IP header and forward; the meaning lives at the ends.

Ignoring header overhead, and sending many tiny packets where a few full ones would do.

Treating the layers as leaky-free. They abstract, but performance problems cross them, which is why you learn all of them.

EXERCISES

EXERCISE 1.1 Place the responsibility

Which layer handles each of these, and why?

- Making sure the bytes arrive in the order they were sent.
 - Choosing which cable or radio to send the next hop over.
 - Deciding which running program on the server receives the data.
 - Finding a route from your city to a server in another country.
-

SOLUTION

- Transport (TCP).** IP delivers packets that may arrive out of order; TCP's sequence numbers restore order at the receiving end (chapter 9).
- Link.** It moves a frame across one physical segment to the next device; it has no idea about the end-to-end path.
- Transport,** via the port number. The IP address gets the packet to the host; the port gets it to the right program (a web server on 443, say).

- d. **Network** (IP), through routing (chapter 14). Each router along the way makes a local next-hop decision based on the destination IP.

EXERCISE 1.2 Do the overhead arithmetic

Using the chapter's header sizes (TCP 20, IP 20, Ethernet 14), a voice-chat app sends 20-byte audio samples, one per packet, 50 times a second. How much of the bandwidth is headers? What would you change?

SOLUTION

Each packet carries 20 bytes of audio under 54 bytes of headers, so $54 / 74 \approx 73\%$ of every packet is overhead, and at 50 packets a second you're paying that 73% tax constantly. (Real voice adds a UDP+RTP header too, making it worse.) The fix is to send fewer, larger packets, bundle several audio samples per packet, trading a little latency for much less overhead, which is exactly the trade every real-time codec tunes. It's also why protocols like QUIC work hard to shrink headers. The deeper point: on the network, the number of packets often matters as much as the number of bytes.

EXERCISE 1.3 Trace the layers

You load a web page and it's slow. A colleague says "the server must be slow". Using the layered model, list four other places the delay could be, one per layer or hop.

SOLUTION

- **Application:** a DNS lookup to resolve the hostname before anything else happens (chapter 15), or a chain of redirects.
- **Transport:** the TCP handshake and TLS handshake, two or more round trips before the first byte of the page (chapters 6, 9), or packet loss triggering retransmission.
- **Network:** a long path, many hops, or congestion queueing your packets, which a traceroute would show (chapter 19).
- **Link / last mile:** a slow or lossy Wi-Fi link, or a congested home connection.

"The server is slow" is one hypothesis out of many, and usually not the right one. The rest of this volume is about telling them apart.

CHECKLIST

- ☐ Can you name the four layers and what each is responsible for?
- ☐ Do you understand encapsulation, each layer wrapping the one above?
- ☐ Do you know which address lives at which layer (hostname, IP, port, MAC)?
- ☐ Can you explain why routers don't read your HTTP?
- ☐ When something's slow, do you think in terms of the whole journey, not just the server?

GO DEEPER

BOOK [Computer Networking: A Top-Down Approach](#) Kurose & Ross · chapter 1, and do the Wireshark labs

BOOK [High Performance Browser Networking](#) Ilya Grigorik · free online

REFERENCE [How the web works](#) MDN

URLs and the web model

3 exercises

The web is a simple idea: every resource has an address, you fetch it by that address over HTTP, and resources link to each other. The address is the URL, and reading one correctly, its scheme, its host and port, its path and query, and knowing what an origin is, is the foundation of everything from routing to security.

Three inventions

Tim Berners-Lee's web rests on three things that still define it: a way to **name** any resource (the URL), a way to **fetch** it (HTTP, chapter 3), and a way to **link** resources into a web (hypertext). This chapter is about the first, because the name comes before the fetch, and because a surprising amount of networking and security is really about reading a URL correctly. The model is deliberately simple: a client asks for the resource at a URL, a server returns a representation of it, and that's a complete interaction. The resource might be a file, a database row rendered as JSON, or something computed on the spot; the URL names it without saying how it's produced.

Anatomy of a URL

EXAMPLE url.py · taking a URL apart

```
from urllib.parse import urlsplit, urljoin, parse_qs, urlencode

DEFAULT_PORTS = {"http": 80, "https": 443, "ws": 80, "wss": 443}

def anatomy(url):
    u = urlsplit(url)
    return {
        "scheme": u.scheme,           # how to talk: https, ws, mailto...
        "host": u.hostname,           # which machine (a name, resolved by DNS)
        "port": u.port or DEFAULT_PORTS.get(u.scheme), # which program; default by scheme
        "path": u.path,               # which resource on that machine
        "query": dict(parse_qs(u.query)), # parameters
        "fragment": u.fragment,       # a spot within the resource; never sent to
                                     the server
    }

def origin(url):
    """The security boundary of the web: scheme + host + port. Same-origin or not."""
    u = urlsplit(url)
    port = u.port or DEFAULT_PORTS.get(u.scheme)
    return f"{u.scheme}://{u.hostname}:{port}"

if __name__ == "__main__":
    url = "https://user@shop.example.com:8443/products/42?color=blue&size=M#reviews"
```

```

for k, v in anatomy(url).items():
    print(f" {k:9}: {v}")

print("\norigins (the web's security boundary):")
for a, b in [("https://app.example.com/a", "https://app.example.com/b"),
             ("https://app.example.com", "http://app.example.com"),      # scheme differs
             ("https://app.example.com", "https://api.example.com")]:    # host differs
    print(f" {origin(a)} vs {origin(b)}: {'SAME' if origin(a) == origin(b) else
        'different'} origin")

print("\nresolving relative URLs against a base (how links in a page work):")
base = "https://docs.example.com/guide/intro.html"
for rel in [ "../setup.html", "../api/ref.html", "/", "//cdn.example.com/x.js", "?v=2" ]:
    print(f" {rel:24} -> {urljoin(base, rel)}")

print("\npercent-encoding keeps special characters out of the structure:")
print(" " + "?" + urlencode({"q": "cats & dogs", "page": "2"}))

```

Each part answers a different question, and they nest from general to specific:

origin = scheme + host + port (the security boundary)

https : // shop.example.com : 8443 /products/42?color=blue&size=M#reviews

scheme	host	port		
protocol	which machine (DNS)	which program		

path · which resource query · parameters fragment · browser only

The authority is host + port. The fragment is never sent to the server; everything else is.

One URL, every part labelled. The origin, the first three parts, is the unit the whole web uses to decide what may talk to what.

A few parts repay attention. The **fragment** (`#reviews`) is handled entirely by the browser and *never sent to the server*, which is why single-page apps once used it for routing and why you can't read it in server logs. The **query** carries parameters, but it's part of the URL, so it ends up in logs, history and referrer headers, never put secrets there (chapter 6 of *The Engineer's Citadel* returns to this). And the term **URI** you'll see in specs is the broader category; for practical purposes a URL is a URI that tells you how to locate the thing, which is the only kind you'll use.

Origin: the web's security boundary

The three parts **scheme**, **host** and **port** together form the **origin**, and the origin is the single most important concept the web uses for security. The browser's same-origin policy keeps one site from reading another's data based on it; cookies, CORS and much else are defined in terms of it (all covered in the security stage). The example shows the rule's sharp edges:

`https` and `http` on the same host are *different* origins, because the scheme differs, and `app.example.com` and `api.example.com` are different origins, because the host differs, even though they're the same site to a human. Getting "same origin" right is getting the scheme, host *and* port all equal.

Relative URLs and links

Most URLs in a page aren't written out in full; they're **relative** to the page's address, and the browser resolves them against a base. The example shows the rules: `./setup.html` replaces the last path segment, `../api/ref.html` climbs a directory, a leading `/` is absolute from the host's root, and a leading `//` is **protocol-relative**, keeping the current scheme but changing the host (handy for CDNs, and a classic open-redirect trap). Relative URLs are what make a site movable between domains without rewriting every link, and understanding resolution is what stops the mysterious "why did this link go *there*" bug.

Percent-encoding and schemes

A URL has structure, `/`, `?`, `&`, `#` all mean something, so any data you put *inside* it that contains those characters must be **percent-encoded** (`%26` for `&`, and so on), exactly as the example encodes "cats & dogs". This is the URL's version of the separate-data-from-structure rule you'll meet everywhere in security: encode data so it can't be mistaken for structure. Beyond `http` and `https`, the scheme can select other protocols entirely, `ws` and `wss` for WebSockets (chapter 7), `mailto:`, `file:`, `data:`, so the scheme is the first thing to check when a URL behaves unexpectedly.

COMMON TRAPS

Putting secrets in the query string, where they leak into logs, history and referrers.

Assuming same host means same origin. Scheme and port count too; `http` and `https` differ.

Expecting the server to see the fragment. It never leaves the browser.

Building URLs by string concatenation, without percent-encoding the data you insert.

Missing protocol-relative `//` links, a common open-redirect and mixed-content source.

EXERCISES

EXERCISE 2.1 Same origin?

The page is at `https://shop.example.com/cart`. For each, same origin or not, and why?

- `https://shop.example.com/checkout`
- `https://shop.example.com:8443/cart`
- `http://shop.example.com/cart`

d. `https://api.shop.example.com/cart`

SOLUTION

- a. **Same origin.** Only the path differs, and the path isn't part of the origin.
- b. **Different.** The port differs (8443 vs the default 443).
- c. **Different.** The scheme differs (http vs https).
- d. **Different.** The host differs (`api.shop` vs `shop`), even though it's a subdomain of the same site. "Same site" and "same origin" are different ideas, which matters for cookies.

EXERCISE 2.2 Resolve the links

A page at `https://site.example/blog/2026/post.html` contains these links. Where does each go?

- a. `image.png`
- b. `/about`
- c. `../2025/old.html`
- d. `//assets.site.example/style.css`

SOLUTION

- a. `https://site.example/blog/2026/image.png` (same directory as the page).
- b. `https://site.example/about` (absolute path from the host root).
- c. `https://site.example/blog/2025/old.html` (up one directory, then down).
- d. `https://assets.site.example/style.css` (protocol-relative: keeps `https` , changes the host).

If you weren't sure, the chapter's `urljoin` example settles it, and it's exactly what the browser does.

EXERCISE 2.3 Build the URL safely

You need to build a search URL for the user's query, which might contain spaces, ampersands or slashes. A colleague writes

`f"https://site.example/search?q={query}"` . What breaks, and what's the fix?

SOLUTION

If the query is `a&b=c`, the raw `&` and `=` are read as URL structure, so the server sees a parameter `q=a` and a bogus parameter `b=c`, not the intended search for `"a&b=c"`. Spaces and slashes cause similar corruption or ambiguity. The fix is to percent-encode the value, which `encodeURIComponent({"q": query})` does, turning it into `q=a%26b%3Dc` so it's carried as data, not structure. This is the same lesson as parameterised queries in a database: never build a structured string by concatenating untrusted data; use the encoder the format provides.

CHECKLIST

- ☐ Can you name every part of a URL and what it's for?
- ☐ Do you know that an origin is scheme *and* host *and* port?
- ☐ Do you understand that the fragment never reaches the server?
- ☐ Do you percent-encode data you put into a URL?
- ☐ Can you resolve a relative URL against a base in your head?

GO DEEPER

REFERENCE [URIs and URLs MDN](#)

SPEC [The URL Standard WHATWG · how browsers really parse URLs](#)

REFERENCE [Same-origin policy MDN](#)

HTTP: the message

3 exercises

HTTP is a request-and-response protocol made of text: a line, some headers, a blank line, and an optional body. Learn its methods, its status codes, and above all how it marks where a message ends, because message framing is what your own server has to get right and where the subtle bugs live.

Text on the wire

An HTTP/1.1 request is simple enough to read and write by hand, which is exactly what this stage's deliverable asks you to do. It's a **request line** (method, target, version), a set of **headers**, a blank line, and an optional **body**. The response has the same shape with a **status line** instead. Parsing and building it is a few lines:

EXAMPLE http.py · parse a request, build a response, decode chunks

```
# HTTP/1.1 is text on the wire: a request line, headers, a blank line, then an optional body.
# This is the heart of the stage's deliverable: parse it yourself, no framework.
def parse_request(raw: bytes):
    head, _, body = raw.partition(b"\r\n\r\n")
    lines = head.split(b"\r\n")
    method, target, version = lines[0].decode().split(" ")
    headers = {}
    for line in lines[1:]:
        name, _, value = line.partition(b":")
        headers[name.decode().strip().lower()] = value.decode().strip() # names are case-insensitive
    return {"method": method, "target": target, "version": version,
            "headers": headers, "body": body}

def build_response(status, reason, headers, body=b''):
    lines = [f"HTTP/1.1 {status} {reason}"]
    headers = {**headers, "Content-Length": str(len(body))} # frame the body by its length
    lines += [f"{k}: {v}" for k, v in headers.items()]
    return ("\r\n".join(lines) + "\r\n\r\n").encode() + body

def decode_chunked(stream: bytes) -> bytes:
    """Chunked transfer-encoding: send a body whose length you don't know up front,
    as a series of (hex size, CRLF, bytes, CRLF) chunks ending with a zero-size chunk."""
    out, i = bytearray(), 0
    while True:
        nl = stream.index(b"\r\n", i)
        size = int(stream[i:nl].split(b";")[0], 16) # hex size, optional ;extension
        if size == 0:
            break
        start = nl + 2
        out += stream[start:start + size]
        i = start + size + 2 # skip the chunk and its trailing CRLF
    return bytes(out)
```

```

if __name__ == "__main__":
    raw = (b"GET /products/42 HTTP/1.1\r\n"
           b"Host: shop.example.com\r\n"
           b"Accept: application/json\r\n"
           b"Connection: keep-alive\r\n"
           b"\r\n")
    req = parse_request(raw)
    print("method:", req["method"], "| target:", req["target"])
    print("host header:", req["headers"]["host"], "| keep-alive:", req["headers"]
          ["connection"])

    resp = build_response(200, "OK", {"Content-Type": "application/json"}, b'{"id":42}')
    print("\nresponse on the wire:")
    print(resp.decode().replace("\r\n", "\\r\\n\\n"))

    chunked = b"9\r\n{\"id\":42}\r\n7\r\n, \"ok\":1\r\n0\r\n\r\n"
    print("decoded chunked body:", decode_chunked(chunked).decode())

```

That small file is most of an HTTP server. The rest of this chapter is what the fields mean and the one thing the parser must never get wrong: framing.

Methods: what you want done

The method states the intent. The ones that matter, and two properties that govern them:

METHOD	MEANS	SAFE?	IDEMPOTENT?
GET	Fetch a resource; no side effects	yes	yes
HEAD	Like GET but headers only, no body	yes	yes
POST	Create, or submit; may have side effects	no	no
PUT	Replace a resource at a known URL	no	yes
PATCH	Partially update	no	no
DELETE	Remove a resource	no	yes

Safe means it only reads, never changes state, so a crawler or a browser prefetch can call it freely (which is why putting a delete behind a GET is a classic bug). **Idempotent** means doing it twice has the same effect as once, which is what lets a client safely retry a timed-out PUT or DELETE but not a POST. These properties aren't enforced by HTTP; they're promises *you* keep when you implement the method, and the whole web (caches, retries, prefetching) assumes you keep them. *The Engineer's Shipyard* builds API design on top of this.

Status codes: how it went

The three-digit code's first digit is the category, and knowing the ranges is most of the job: **1xx** informational (rare), **2xx** success, **3xx** redirection ("look elsewhere"), **4xx** you (the client) got it wrong, **5xx** I (the server) got it wrong. The ones worth memorising: **200 OK**, **201 Created**, **204 No Content**; **301** / **308** permanent and **302** / **307** temporary redirects, **304 Not Modified** (your cached copy is still good, chapter 4); **400** bad request, **401** unauthenticated, **403** forbidden, **404** not found, **409** conflict, **429** too many requests; **500** internal error, **502** bad gateway, **503** unavailable, **504** gateway timeout. The 4xx-versus-5xx line is the one that matters operationally: a spike in 4xx is usually a client or a bad deploy; a spike in 5xx is you.

Framing: where does the message end?

This is the question HTTP exists to answer on a byte stream, and the one your server must get exactly right. TCP gives you a stream of bytes with no message boundaries (chapter 9), so HTTP has to say where each message's body stops. There are two ways. **Content-Length** states the body's size up front, and the receiver reads exactly that many bytes, what `build_response` does. But sometimes you don't know the size when you start sending (a response generated on the fly), so **chunked transfer encoding** sends the body as a series of sized chunks, each prefixed with its length in hex, ending with a zero-length chunk, which `decode_chunked` reassembles. Every HTTP/1.1 body uses one or the other. Getting framing wrong isn't just a bug; when a proxy and a server disagree about where a message ends, you get request smuggling, one of the nastier web vulnerabilities (*The Engineer's Citadel*, chapter 6), which is why the parser's handling of Content-Length and chunked has to be strict.

Keep-alive: reuse the connection

Opening a TCP connection costs a round trip (chapter 9), and HTTPS adds another for TLS (chapter 6), so opening a fresh connection for every request, as HTTP/1.0 did, is slow. HTTP/1.1 made connections **persistent** by default: after a response, the connection stays open (`Connection: keep-alive`) and the next request reuses it, paying the handshake once.

WITHOUT KEEP-ALIVE

H   req/resp H   req/resp H   req/resp
handshake per request • 3 extra round trips

WITH KEEP-ALIVE (HTTP/1.1 default)

H   req/resp  req/resp  req/resp
one handshake, then reuse • 2 round trips saved

 = TCP handshake (1 round trip), H

Still one-at-a-time: request 2 waits for response 1 (head-of-line). HTTP/2 fixes that.

Persistent connections pay the handshake once instead of per request. But over HTTP/1.1 the requests still run one at a time on the connection, which is the limit chapter 5's HTTP/2 removes.

One subtlety the diagram flags: even with keep-alive, HTTP/1.1 sends requests *serially* on a connection, the next can't start until the previous response finishes (an attempt to overlap them, called pipelining, failed in practice because of head-of-line blocking). Browsers worked around it by opening six connections per host; HTTP/2 (chapter 5) solves it properly.

One more essential header: Host

HTTP/1.1 made the `Host` header mandatory, and it's why the modern web works: one server at one IP address can host thousands of sites, and the `Host` header tells it which one you want (**virtual hosting**). Without it, the IP address alone couldn't distinguish `a.example` from `b.example` on the same machine. It's also the header to be careful with on the server side, since trusting it blindly causes the cache-poisoning and password-reset bugs from *The Engineer's Citadel*.

COMMON TRAPS

Side effects behind GET, which crawlers, prefetchers and caches will trigger.

Retrying a POST as if it were idempotent, causing duplicate charges or records.

Getting framing wrong: trusting both Content-Length and chunked, or miscounting bytes, the door to request smuggling.

Returning 200 with an error in the body, so clients, caches and monitoring can't tell success from failure.

Opening a new connection per request, paying the handshake every time.

EXERCISES

EXERCISE 3.1 Choose the method and status

For each, give the HTTP method and the success status code.

- Create a new order and return it.
- Fetch an order.
- Replace an order's shipping address at a known URL.
- Delete an order, returning nothing.

SOLUTION

- a. **POST** /orders → **201 Created**, with a `Location` header pointing to the new order.
- b. **GET** /orders/42 → **200 OK**.
- c. **PUT** /orders/42/address → **200 OK** (or 204). PUT because you're replacing at a known URL, and it's idempotent, so a retry is safe.
- d. **DELETE** /orders/42 → **204 No Content**. Idempotent: deleting an already-deleted order can still return 204 (or 404).

EXERCISE 3.2 Frame the response

Your socket server streams a large report it generates row by row, so it doesn't know the total size when the response starts. How do you frame the body, and what does the client see on the wire?

SOLUTION

Use **chunked transfer-encoding**: send `Transfer-Encoding: chunked` instead of `Content-Length`, then emit each batch of rows as a chunk, the size in hex, a CRLF, the bytes, a CRLF, and finish with a `0`-size chunk and a final CRLF. The client reads chunks until the zero-size one, reassembling the body exactly as the chapter's `decode_chunked` does. This lets you start sending before you've finished generating, which is the whole point, and it's one of the two framing methods your deliverable must implement. Don't send both `Content-Length` and `Transfer-Encoding`; a receiver that honours one while a proxy honours the other is the request-smuggling hole.

EXERCISE 3.3 Diagnose the status codes

Your dashboard shows these shifts after a deploy. What does each most likely mean?

- a. A sudden wave of 500s.
- b. A wave of 404s on URLs that worked yesterday.
- c. A rise in 429s.
- d. Lots of 502s and 504s.

SOLUTION

- a. **500s**: your code is throwing, a bug in the new deploy. Roll back and check logs.

- b. **404s on formerly-good URLs:** a routing or deploy problem, a renamed path, a broken rewrite, a bad static-asset build, not the users' fault.
- c. **429s:** clients are being rate-limited, maybe a retry storm, a misbehaving client, or limits set too low.
- d. **502/504:** the gateway (a load balancer or proxy) can't reach or isn't getting timely answers from your backend, the backend is down, overloaded, or slow. The distinction from 500 tells you the failure is *between* the proxy and the server, not in the app's own logic.

CHECKLIST

- ☐ Can you read and write a raw HTTP/1.1 request and response?
- ☐ Do you know which methods are safe and which are idempotent, and keep those promises?
- ☐ Do you use status codes correctly, especially the 4xx-versus-5xx line?
- ☐ Does your server frame bodies correctly, by Content-Length or chunked, never ambiguously?
- ☐ Do you reuse connections with keep-alive rather than reconnecting per request?

GO DEEPER

REFERENCE [MDN HTTP reference](#) the reference you'll return to for years

SPEC [RFC 9110: HTTP Semantics IETF](#)

BOOK [Computer Networking: A Top-Down Approach](#) Kurose & Ross · the application-layer chapter

HTTP: caching, cookies and negotiation

3 exercises

HTTP is stateless by design: every request stands alone, which is what lets the web scale. On that foundation it adds three things that make it practical, caching so you don't fetch what you already have, cookies so a stateless protocol can remember you, and negotiation so one URL can serve the right format to each client.

Stateless by design

HTTP keeps no memory between requests: the server handles each one with no idea what you asked a moment ago. That sounds like a limitation, and it's actually the web's superpower, because any server can handle any request, so you can put a thousand servers behind a load balancer and it just works (chapter 18). The cost is that anything that *should* persist, who you are, what's in your cache, has to be carried explicitly, which is what this chapter's three mechanisms do.

Caching: the biggest lever there is

The fastest request is the one you never send. HTTP caching lets a response be reused, by the browser, a CDN, or a proxy in between, instead of fetched again, and getting it right is the single biggest performance win on the web. Two ideas drive it. **Freshness**: a response can carry `Cache-Control: max-age=600`, meaning "reuse me for 600 seconds without asking". **Validation**: when a cached copy goes stale, the cache doesn't blindly refetch; it asks "is my copy still good?" with the `ETag` it was given, and the server answers `304 Not Modified` (keep it, a tiny response with no body) or sends a fresh `200`. Here is the whole decision:

EXAMPLE `cache.py` · fresh, revalidate, or fetch

```
# An HTTP cache (browser, CDN or proxy) decides per request: serve a fresh copy, revalidate
# a stale one cheaply, or fetch anew. The rules come from Cache-Control and ETag headers.
class Origin:
    """A stand-in web server. The resource changes only when we say so."""
    def __init__(self):
        self.body, self.etag, self.max_age = b"v1 body", "v1", 10
    def fetch(self, if_none_match=None):
        if if_none_match == self.etag:                                # client's copy is still current
            return 304, None, self.etag, self.max_age                # tiny response, no body
        return 200, self.body, self.etag, self.max_age

class Cache:
    def __init__(self):
        self.store = {}      # url -> (body, etag, stored_at, max_age)

    def get(self, url, origin, now):
        entry = self.store.get(url)
```

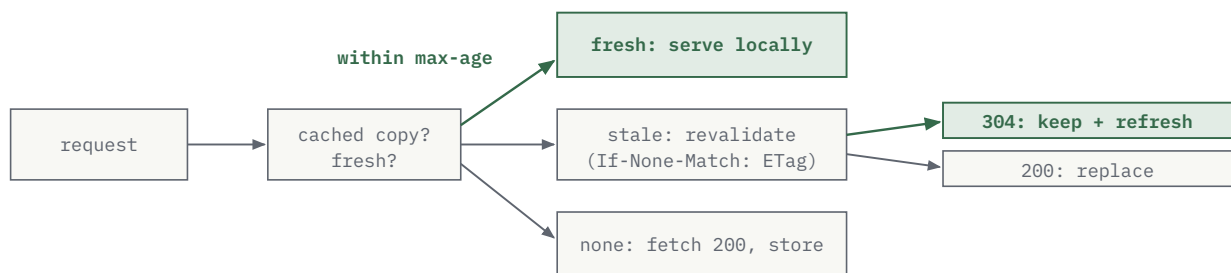
```

if entry is None:
    status, body, etag, max_age = origin.fetch()
    self.store[url] = (body, etag, now, max_age)
    return "MISS -> fetched", body
body, etag, stored_at, max_age = entry
if now - stored_at < max_age:
    return "HIT (fresh)", body # no network at all
# Stale: revalidate cheaply with the ETag we hold.
status, new_body, new_etag, new_max_age = origin.fetch(if_none_match=etag)
if status == 304:
    self.store[url] = (body, etag, now, new_max_age) # still good; refresh freshness
    return "REVALIDATE -> 304 (kept)", body
self.store[url] = (new_body, new_etag, now, new_max_age)
return "REVALIDATE -> 200 (replaced)", new_body

if __name__ == "__main__":
    cache, origin, url = Cache(), Origin(), "https://shop.example/logo.png"
    for now in (0, 5, 12, 20):
        label, body = cache.get(url, origin, now)
        print(f"t={now:2}s {label:28} body={body.decode()}")
    print("\nnow the resource changes on the origin:")
    origin.body, origin.etag = b"v2 body", "v2"
    label, body = cache.get(url, origin, now=35) # stale -> revalidate -> 200
    print(f"t=35s {label:28} body={body.decode()}")

```

The trace shows it: a miss fetches and stores; within `max-age` it's a pure hit with no network; once stale it revalidates with the ETag and gets a `304`, so it keeps the bytes it already had and just refreshes the freshness clock; and when the resource actually changes, revalidation returns `200` with the new body. That `304` is the quiet workhorse of the web: it turns "re-download the image" into "a few bytes confirming the image is unchanged".



The cache's choice on every request. Most hits never touch the network; most "misses" on stale content are cheap 304s, not full downloads. Cache-Control and ETag are the two knobs that drive it.

A few more knobs matter in practice. `no-store` forbids caching entirely (for private, per-user data); `no-cache` confusingly means "cache it but always revalidate before use"; `private` lets the browser cache but not a shared CDN; `s-maxage` sets a separate lifetime for shared caches. And the `Vary` header tells caches which request headers change the response (`Vary: Accept-Encoding`), so they don't hand a gzipped copy to a client that can't

decompress it. The pattern that makes static assets both cacheable forever and instantly updatable is to put a content hash in the filename (`app.9f3c.js`) and serve it with a one-year `max-age` : the URL changes when the content does.

Cookies: memory for a stateless protocol

To recognise you across requests, the server sends `Set-Cookie: session=abc123` with a response, the browser stores it, and sends `Cookie: session=abc123` on every subsequent request to that site. That round trip is how a stateless protocol gets state: the cookie is a little token the client carries back each time, and the server looks up who it belongs to. Cookies carry attributes that control their safety and scope, `Secure` (HTTPS only), `HttpOnly` (hidden from JavaScript), `SameSite` (not sent on cross-site requests), `Domain` and `Path` (scope), and an expiry. The security of these is a whole topic in its own right (*The Engineer's Citadel*, chapter 5); here the point is the mechanism: state on the web is something the client carries, not something the connection remembers.

Content negotiation: one URL, many representations

The same resource can come in different forms, and the client says which it prefers through `Accept` headers, while the server picks and announces its choice with `Content-Type` . The main axes: **type** (`Accept: application/json` vs `text/html`), **language** (`Accept-Language: fr`), and **encoding** (`Accept-Encoding: gzip, br, zstd`). That last one is quietly one of the web's biggest wins: the client advertises which **compression** it understands, the server compresses the response, and text payloads shrink by 70% or more over the wire, decompressed transparently by the browser. Whenever the response depends on one of these headers, the server must set `Vary` so caches key on it, the classic bug being a cache that serves a French page to an English visitor because it ignored `Vary: Accept-Language` .

COMMON TRAPS

Caching per-user data in a shared cache, leaking one user's response to another, forgetting `private` or `no-store` .

No cache headers at all, so everything is refetched, or browsers guess.

Caching by URL while ignoring `Vary` , serving the wrong language or a gzipped body to a client that can't read it.

Confusing `no-cache` (revalidate) with `no-store` (never cache).

Not compressing text responses, leaving an easy 70% off the wire unclaimed.

EXERCISES

EXERCISE 4.1 Set the cache headers

Choose `Cache-Control` (and any other cache headers) for each.

- A hashed static asset, `app.9f3c.js`.
- A logged-in user's account page.
- The home page, which changes a few times a day and is the same for everyone.
- A bank balance API response.

SOLUTION

- `Cache-Control: public, max-age=31536000, immutable`. The content hash is in the URL, so it can be cached forever; a new build gets a new URL.
- `Cache-Control: private, no-cache` (plus an ETag). The browser may cache it, but a shared CDN must not, and it should revalidate before each use so a logout or change is seen.
- `Cache-Control: public, max-age=300` with an ETag, perhaps `s-maxage` longer for the CDN with purging on publish. Short freshness plus cheap revalidation.
- `Cache-Control: no-store`. Sensitive, must never be written to any cache.

EXERCISE 4.2 Trace the revalidation

Using the chapter's cache with `max-age=10`, a client requests the same image at $t=0$, $t=8$, and $t=15$, and the image hasn't changed. What happens at each, and how many bytes of body cross the network in total (body is 100 KB, a 304 is ~200 bytes)?

SOLUTION

$t=0$: miss, fetch 100 KB, store. **$t=8$:** still within max-age ($8 < 10$), a pure hit, no network. **$t=15$:** stale, so revalidate with the ETag; the image is unchanged, so the server returns `304`, ~200 bytes, and the cache keeps its copy and refreshes freshness. Total body over the network: **100 KB once**, plus a ~200-byte 304. Without validation you'd have refetched the full 100 KB at $t=15$; the ETag turned that into 200 bytes. Multiply by every image on every page view and the saving is the web's performance, much of it invisible, running on 304s.

EXERCISE 4.3 Find the caching bug

Users report occasionally seeing *another* user's name on their dashboard, but only when behind a corporate proxy. The dashboard is served with `Cache-Control: max-age=60`. What's wrong?

SOLUTION

The response is per-user but marked `max-age=60` with no `private`, so the **shared** corporate proxy cache is allowed to store one user's personalised dashboard and serve it to the next user who requests the same URL within 60 seconds. That's why it only happens behind the shared proxy and not for home users (whose browser cache is private to them). The fix is `Cache-Control: private, no-store` (or at least `private`) on anything personalised, so no shared cache ever holds it. This is a real and serious class of bug, a caching misconfiguration that becomes an information-disclosure vulnerability, which is why cache directives on authenticated responses deserve care.

CHECKLIST

- ☐ Does every response carry deliberate cache headers, not whatever the framework defaults to?
- ☐ Is per-user data marked `private` or `no-store` so shared caches can't hold it?
- ☐ Do you use ETags so stale content revalidates cheaply with 304s?
- ☐ Do you set `Vary` whenever the response depends on a request header?
- ☐ Are text responses compressed, and do you understand cookies as client-carried state?

GO DEEPER

REFERENCE [HTTP caching MDN](#)

GUIDE [RFC 9111: HTTP Caching IETF](#)

REFERENCE [Using HTTP cookies MDN](#)

HTTP/2 and HTTP/3

3 exercises

The HTTP you write (chapters 3 and 4) is the same across versions; what changed underneath is how messages share a connection. HTTP/2 multiplexes many requests over one TCP connection and HTTP/3 moves that onto QUIC to remove the last head-of-line blocking. Knowing the difference tells you why a page is fast, and why a lossy link makes one version crawl.

The problem each version solves

HTTP/1.1 sends requests one at a time per connection (chapter 3), so a page needing fifty resources is slow. Browsers worked around it by opening about six connections per host, but that's a hack: it's limited, and each connection pays its own handshake. The real fix is to carry many requests on one connection at once, which is **multiplexing**, and the story of HTTP/2 and HTTP/3 is multiplexing done progressively better.

HTTP/2: many streams, one TCP connection

HTTP/2 keeps HTTP's meaning identical but changes the wire format from text to **binary frames**, and lets many **streams** share a single TCP connection, each stream an independent request/response, interleaved frame by frame. One connection now carries all fifty resources concurrently. It adds **header compression** (HPACK), because repetitive headers (cookies, user-agent) on every request waste bandwidth, and stream prioritisation. The result is a big speed-up over 1.1 for typical pages, and it's why HTTP/2 reached about 80% of top sites.

But HTTP/2 has a flaw it inherited from TCP. All those streams ride one TCP connection, and **TCP delivers bytes strictly in order**: if one packet is lost, TCP holds back everything that arrived after it until the missing packet is retransmitted, a round trip later. So a single lost packet stalls *every* stream on the connection, even streams whose own data arrived fine. This is **transport-level head-of-line blocking**, and on a lossy link (mobile, congested Wi-Fi) it can make HTTP/2 slower than HTTP/1.1's six separate connections, because with six connections a loss only stalls one of them.

HTTP/3: multiplexing on QUIC

HTTP/3 fixes this by abandoning TCP. It runs over **QUIC** (RFC 9000), a transport built on UDP that provides its own streams, and crucially QUIC's streams are *independent*: a lost packet blocks only the stream whose data it carried, while every other stream keeps flowing. The head-of-line blocking is gone. QUIC brings three more wins because it was designed from scratch: the transport and TLS handshakes are **combined** into one round trip (and 0 round trips on resumption), so a fresh HTTPS connection is faster; it supports

connection migration, so your download survives switching from Wi-Fi to cellular without starting over; and encryption is built in, not bolted on. The cost is that it's UDP, which some networks throttle or block, so clients fall back to HTTP/2 when QUIC can't get through.

EXAMPLE hol.py · where each version wins and loses

```
import math

# A simplified model of fetching N resources, to show where each HTTP version wins and loses.
# One "round" is one round-trip time (RTT). Not packet-accurate, but it captures the mechanism.

def http11_rounds(n, connections=6):
    """HTTP/1.1: serial per connection; browsers open ~6 connections per host."""
    return math.ceil(n / connections)          # each connection does its share
        serially

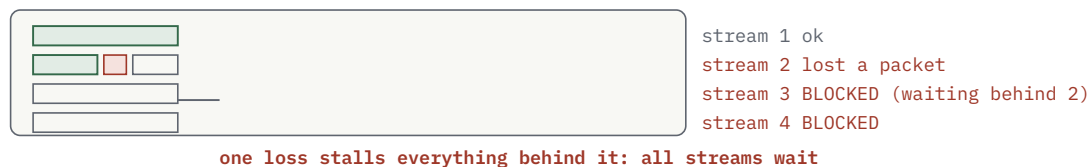
def multiplexed(n, rtt_ms, lost_stream=True):
    """HTTP/2 and HTTP/3 both put all N streams in flight at once: ~1 round with no loss.
    The difference is what one lost packet does."""
    base = rtt_ms
    # One packet is lost mid-transfer; recovering it costs ~1 extra RTT.
    h2_completion = base + rtt_ms               # TCP delivers in order, so the gap stalls EVERY
        stream
    h2_delayed = n
    h3_completion = base                       # QUIC streams are independent...
    h3_one_stream = base + rtt_ms              # ...only the stream on the lost packet waits
    h3_delayed = 1
    return {
        "h2": {"all_streams_done_at_ms": h2_completion, "streams_delayed": h2_delayed},
        "h3": {"most_done_at_ms": h3_completion, "one_stream_at_ms": h3_one_stream,
            "streams_delayed": h3_delayed},
    }

if __name__ == "__main__":
    n, rtt = 20, 50
    print(f"fetching {n} resources, RTT {rtt}ms:\n")
    print(f"HTTP/1.1 (6 connections): {http11_rounds(n)} rounds = {http11_rounds(n) * rtt}ms"
        f" (serial per connection)")
    print(f"HTTP/2 or HTTP/3 (no loss): 1 round = {rtt}ms (all {n} streams multiplexed)\n")

    print("now one packet is lost mid-transfer:")
    r = multiplexed(n, rtt)
    print(f" HTTP/2 (one TCP pipe): all {r['h2']['streams_delayed']} streams wait -> "
        f"{r['h2']['all_streams_done_at_ms']}ms (TCP head-of-line blocking)")
    print(f" HTTP/3 (QUIC streams): {n - r['h3']['streams_delayed']} streams done at "
        f"{r['h3']['most_done_at_ms']}ms, only {r['h3']['streams_delayed']} waits to "
        f"{r['h3']['one_stream_at_ms']}ms")
```

The model shows the whole arc. With no loss, HTTP/2 and HTTP/3 both crush HTTP/1.1 (one round versus four) by multiplexing. Under a single packet loss, HTTP/2's shared TCP pipe stalls all twenty streams while HTTP/3 delays only the one stream that lost a packet. That difference is the entire reason HTTP/3 exists.

HTTP/2 · one TCP pipe, in-order



HTTP/3 · independent QUIC streams



The same packet loss, two transports. On HTTP/2 one gap in the single ordered TCP stream holds back everything behind it; on HTTP/3 each QUIC stream is delivered independently, so the loss is contained.

What this means for you

Mostly, you turn it on. HTTP/2 and HTTP/3 are enabled at the server, load balancer or CDN, and because the *semantics* are unchanged, your application code and the headers of chapters 3 and 4 work as before. Two practical consequences, though. Old HTTP/1.1 performance hacks become counter-productive on HTTP/2: concatenating many files into one, "domain sharding" across hostnames, and inlining assets all fight multiplexing, so drop them. And protocol features are attack surface: in 2023 the **HTTP/2 Rapid Reset** attack (CVE-2023-44487) abused the cheapness of opening and instantly cancelling a stream to generate record-breaking denial-of-service floods, up to 398 million requests a second, a reminder that every new capability is also a new way to be attacked.

COMMON TRAPS

Keeping HTTP/1.1 hacks (file concatenation, domain sharding, inlining) on HTTP/2, where they hurt.

Assuming HTTP/2 is always faster. On a lossy link its TCP head-of-line blocking can lose to 1.1.

Forgetting HTTP/3 needs UDP, which some networks block, so a fallback path must exist.

Thinking a new version changes HTTP's meaning. Methods, status codes and headers are the same.

Ignoring that multiplexing concentrates risk onto one connection and one protocol's quirks.

EXERCISES

EXERCISE 5.1 Explain the mobile mystery

A page loads fast on your office Wi-Fi over HTTP/2 but is noticeably slower on a train, even when the connection has decent bandwidth. The site doesn't support HTTP/3. What's happening, and what would help?

SOLUTION

A train's mobile connection has **packet loss**, not just lower bandwidth, and HTTP/2 puts every stream on one TCP connection. Each lost packet triggers TCP head-of-line blocking: all your concurrent streams stall for a retransmission round trip, so the page stutters even though raw bandwidth is fine. What helps: **HTTP/3**, whose independent QUIC streams contain the loss to the one affected stream, which is exactly why it was designed and why it shines on mobile. (Ironically, old-style six-connection HTTP/1.1 would also localise the loss better than HTTP/2 here, which is the trap in exercise-style thinking that "newer is always faster".)

EXERCISE 5.2 Audit the old optimisations

A site migrating to HTTP/2 still does these HTTP/1.1-era tricks. For each, keep or drop?

- Bundling 30 small JavaScript modules into one big file.
- Serving images from `img1.site`, `img2.site`, `img3.site` (domain sharding).
- Inlining a small critical CSS file into the HTML.
- Using a CDN.

SOLUTION

- Mostly drop.** HTTP/2 multiplexes, so many small files are cheap to request, and separate files cache and update independently (a change to one module doesn't bust the whole bundle). Some bundling still helps compression and reduces per-file overhead, so it's a balance, not a hard rule.
- Drop.** Sharding across hostnames forces multiple connections and defeats multiplexing (and each new host needs its own DNS lookup, connection and handshake). One connection is the goal now.

- c. **Usually drop** for large or shared CSS (it can't be cached separately), though inlining a tiny amount of truly critical CSS to avoid a round trip on first paint can still be worth it. Measure (chapter 16 of the AI volume's cousin, web perf).
- d. **Keep.** A CDN is orthogonal and even more valuable, and it's usually where you enable HTTP/2 and HTTP/3 in the first place (chapter 17).

EXERCISE 5.3 Reason about QUIC's extras

Beyond fixing head-of-line blocking, name two things QUIC does that TCP+TLS can't, and why each matters.

SOLUTION

- **Combined handshake:** QUIC merges the transport and TLS 1.3 handshakes into one round trip (zero on resumption), where TCP+TLS needs the TCP handshake *then* the TLS handshake, two round trips before any data. On a high-latency link that's a visible saving on every new connection.
- **Connection migration:** a QUIC connection is identified by a connection ID, not the IP-address-and-port four-tuple, so it survives a network change, walking out of Wi-Fi onto cellular, without dropping and re-establishing. A TCP connection is bound to the four-tuple and breaks when your IP changes, forcing a reconnect. This matters enormously for mobile.

Both come from QUIC living in user space over UDP, free of TCP's decades of ossified assumptions, which is the deeper reason a new transport was worth the effort.

CHECKLIST

- ☐ Can you explain multiplexing and why it beats HTTP/1.1's serial connections?
- ☐ Do you understand TCP head-of-line blocking, and why HTTP/2 suffers it but HTTP/3 doesn't?
- ☐ Do you know HTTP/3 runs on QUIC over UDP, with a combined handshake and connection migration?
- ☐ Have you removed HTTP/1.1-era hacks that fight multiplexing?

☐ Is there a fallback to HTTP/2 when QUIC is blocked?

GO DEEPER

BOOK High Performance Browser Networking: HTTP/2 Ilya Grigorik

SPEC RFC 9114: HTTP/3 and RFC 9000: QUIC IETF

REFERENCE HTTP evolution MDN

PART II

Secure and real-time

Three chapters on what sits beside plain request-and-response: the TLS layer that makes it private and trustworthy, the transports that keep a connection open for real-time data, and the structured calls of RPC.

TLS and HTTPS

3 exercises

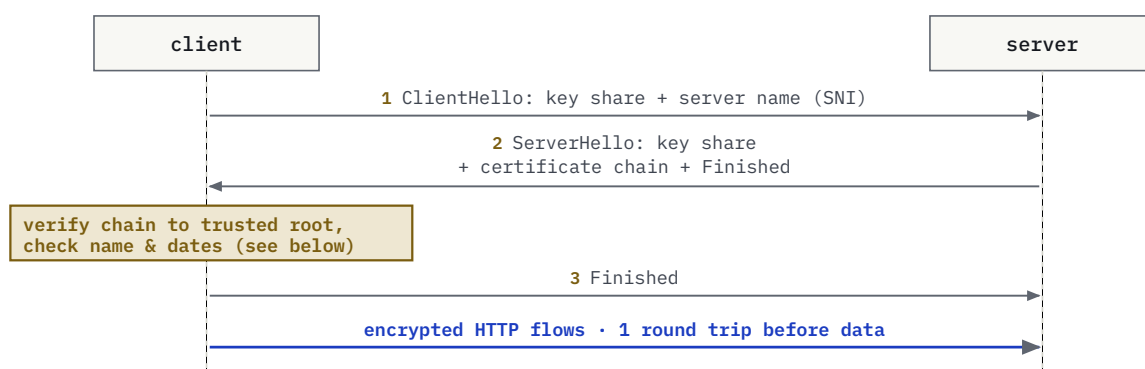
HTTPS is HTTP wrapped in TLS, which gives three things at once: no one can read the traffic, no one can tamper with it undetected, and you know you're really talking to the site you asked for. The first two come from encryption; the third, the hard one, comes from certificates and a chain of trust that ends at a root your device already trusts.

What TLS provides

TLS (Transport Layer Security, the successor to SSL) sits between TCP and HTTP and turns a plain, readable connection into a secure one. It provides **confidentiality**, the bytes are encrypted, so a router, a Wi-Fi snooper or your ISP sees only ciphertext; **integrity**, any tampering is detected and the connection fails rather than delivering altered data; and **authentication**, you can be sure the server is who it claims to be. The first two are "just" encryption. The third is the part that makes the web trustworthy and the part most worth understanding, because encrypting a conversation with an impostor buys you nothing.

The handshake

Before any HTTP flows, client and server run a handshake to agree on keys and verify identity. TLS 1.3, the current version, does it in a single round trip:



TLS 1.3 in one round trip: the client and server exchange key shares (deriving a shared secret neither sent on the wire), the server proves its identity with a certificate, and data flows encrypted. On a resumed connection, 0-RTT can send data in the very first packet.

Two things are happening at once. A **key exchange** (Diffie-Hellman) lets both sides derive the same secret key without ever sending it, using fresh, ephemeral keys per connection so that stealing the server's long-term key later can't decrypt past traffic (**forward secrecy**).

And **authentication**: the server sends its certificate chain and signs the handshake, proving it holds the private key for the certificate. Connecting to a real server shows the result:

EXAMPLE terminal · openssl against a live host (illustrative)

```
$ echo | openssl s_client -connect example.com:443 -servername example.com -brief
Protocol version: TLSv1.3
Ciphersuite: TLS_AES_256_GCM_SHA384
Peer certificate: CN=example.com
Signature type: ecdsa_secp256r1_sha256
Verification: OK
```

Certificates and the chain of trust

A **certificate** binds a public key to a name (`example.com`), and is signed by a **certificate authority** (CA). You don't trust the certificate because it says so; you trust it because you can follow its signatures up a **chain** to a root CA your device already trusts. Your operating system and browser ship with a list of a few hundred trusted root CAs, and validation means: the name matches the site, the dates are current, each certificate was really signed by the next one up, and the top is a trusted root. A real chain looks like this:

EXAMPLE terminal · the chain for example.com (illustrative)

```
subject= CN=example.com                <- leaf (the site)
issuer = Cloudflare TLS Issuing ECC CA 3
subject= Cloudflare TLS Issuing ECC CA 3 <- intermediate
issuer = SSL.com TLS Transit ECC CA R2
subject= SSL.com TLS ECC Root CA 2022   <- root (in your trust store)
```

And the validation logic, which every browser runs on every connection, is only this:

EXAMPLE chain.py · verifying a chain of trust

```
from dataclasses import dataclass

# A certificate says "this public key belongs to this name", signed by an issuer. Your browser
# trusts a leaf only if it can follow the signatures up to a root CA it already trusts.
@dataclass
class Cert:
    subject: str      # who this cert is for
    issuer: str        # who signed it (the subject of the cert above)
    signed_by: str     # key id that actually signed it (must match the issuer's key)
    key_id: str        # this cert's own key id
    not_before: int
    not_after: int
    is_ca: bool = False

TRUSTED_ROOT_KEYS = {"root-key-ssl2022"} # the browser/OS ships a list of these

def verify_chain(chain, hostname, now):
    """chain = [leaf, intermediate(s) ..., root]. Returns (ok, reason)."""
```

```

leaf = chain[0]
if leaf.subject != hostname:
    return False, f"name mismatch: cert is for {leaf.subject}, not {hostname}"
for i, cert in enumerate(chain):
    if not (cert.not_before <= now <= cert.not_after):
        return False, f"{cert.subject}: outside validity window"
    if i + 1 < len(chain):
        issuer = chain[i + 1]
        if cert.issuer != issuer.subject:
            return False, f"{cert.subject}: issuer name doesn't match next cert"
        if cert.signed_by != issuer.key_id:
            return False, f"{cert.subject}: signature not from the next cert's key"
        if not issuer.is_ca:
            return False, f"{issuer.subject} signed a cert but is not a CA"
root = chain[-1]
if root.key_id not in TRUSTED_ROOT_KEYS:
    return False, f"chain ends at {root.subject}, not a trusted root"
return True, "trusted"

if __name__ == "__main__":
    # each cert's signed_by must equal the key_id of the cert above it (the real signature)
    leaf = Cert("example.com", "Issuing CA", signed_by="k-issuing", key_id="k-
        leaf", not_before=0, not_after=200)
    inter = Cert("Issuing CA", "Root CA 2022", signed_by="root-key-ssl2022", key_id="k-
        issuing", not_before=0, not_after=3650, is_ca=True)
    root = Cert("Root CA 2022", "Root CA 2022", signed_by="root-key-ssl2022", key_id="root-
        key-ssl2022", not_before=0, not_after=7300, is_ca=True)
    chain = [leaf, inter, root]

    print("valid chain:      ", verify_chain(chain, "example.com", now=100))
    print("wrong hostname:   ", verify_chain(chain, "evil.com", now=100))
    print("expired leaf:      ", verify_chain(chain, "example.com", now=250))
    forged = [Cert("example.com", "Issuing CA", signed_by="k-attacker", key_id="k-leaf",
        not_before=0, not_after=200), inter, root]
    print("forged signature: ", verify_chain(forged, "example.com", now=100))
    # a fully self-consistent chain whose root simply isn't in the trust store
    rogue_root = Cert("Rogue Root", "Rogue Root", signed_by="k-rogue", key_id="k-rogue",
        not_before=0, not_after=7300, is_ca=True)
    rogue_int = Cert("Issuing CA", "Rogue Root", signed_by="k-rogue", key_id="k-issuing",
        not_before=0, not_after=3650, is_ca=True)
    print("untrusted root:   ", verify_chain([leaf, rogue_int, rogue_root], "example.com",
        now=100))

```

The checks are exactly the ones an attacker must defeat, and the example shows each failing safely: a certificate for the wrong name, an expired one, one whose signature doesn't come from its stated issuer, and a chain that ends at a root nobody trusts. This is the whole security of HTTPS: anyone can generate a certificate claiming to be your bank, but they can't make it *chain to a trusted root* without a CA signing it. The weakness, that *any* of the hundreds of CAs can vouch for *any* name, is real, and the defences against a misbehaving CA (Certificate Transparency, pinning, shrinking lifetimes) are the subject of the security stage.

SNI, and serving many sites

One server at one IP address can host thousands of HTTPS sites, just as with the `Host` header (chapter 3), but there's a chicken-and-egg problem: the server must present the *right certificate* before the encrypted connection exists, so it can't read a `Host` header yet. The

solution is **Server Name Indication (SNI)**: the client puts the hostname in the `ClientHello`, in the clear, so the server knows which certificate to send. That cleartext hostname is the one piece of a TLS connection an eavesdropper can still see, which is why **Encrypted Client Hello (ECH)**, standardised in 2026, now encrypts the SNI too, closing the last metadata leak.

Mutual TLS, and HTTPS everywhere

Normally only the server proves its identity. With **mutual TLS (mTLS)** the *client* presents a certificate as well, so both sides are authenticated, which is how services inside a system authenticate to each other without passwords (the zero-trust model of the production and security stages). And the web has moved to HTTPS by default: browsers mark plain HTTP as insecure, **HSTS** lets a site insist browsers only ever connect over HTTPS, and certificates are now free (via Let's Encrypt) and short-lived, which means the one rule you must follow is to **automate renewal** with ACME, because a manually-renewed certificate is an outage waiting for a date.

COMMON TRAPS

Thinking encryption alone makes you safe. Without certificate validation you may have a private channel to an impostor.

Disabling certificate verification "to make it work", in a client or a test that reaches production.

Manually renewed certificates, which expire and take the site down.

Forgetting the extra round trip: a fresh HTTPS connection costs TCP + TLS handshakes before any data.

Assuming TLS hides everything. The SNI (without ECH) and the destination IP are still visible.

EXERCISES

EXERCISE 6.1 Name the guarantee

Which TLS guarantee, confidentiality, integrity, or authentication, does each scenario rely on?

- Your password isn't readable by the coffee-shop Wi-Fi you're on.
- An ISP can't silently inject ads into a page you load.
- You're sure the login page is really your bank's, not a look-alike on the same network.

SOLUTION

- a. **Confidentiality.** Encryption means the Wi-Fi sees ciphertext.
- b. **Integrity.** TLS detects modification, so injected content breaks the connection rather than reaching you, this is why ISP ad-injection died with HTTPS.
- c. **Authentication.** The certificate chain proves the server holds a certificate for your bank's name, signed by a trusted CA; a look-alike can't produce one. Encryption without this (a) and (b) would still let you have a perfectly private conversation with the attacker.

EXERCISE 6.2 Why did validation fail?

Using the chapter's `verify_chain`, a connection to `shop.example` fails. For each symptom, which check caught it?

- a. The certificate is for `*.shop.example` but presented for `shop.example`'s API at `api.other.example`.
- b. The site's certificate expired yesterday.
- c. The browser says "issued by an unknown authority".

SOLUTION

- a. The **name match** check: the certificate's subject doesn't cover `api.other.example`. (Wildcards match one label, so even `*.shop.example` wouldn't cover a different domain.)
- b. The **validity window** check: `now` is past `not_after`. This is the "forgot to renew" outage, now mitigated by automation.
- c. The **trusted root** check: the chain ends at a root not in the device's trust store, a self-signed or private CA, or a chain the server failed to send completely. Browsers can't follow it to something they trust.

EXERCISE 6.3 Count the round trips

A user on a connection with 80ms RTT loads `https://new-site.example` for the first time (not cached, not previously connected). Count the round trips before the first byte of HTML arrives, and where each goes.

SOLUTION

- **DNS** (chapter 15): ~1 round trip to resolve the hostname to an IP (often local-cache fast, but cold here).
- **TCP handshake** (chapter 9): 1 round trip (SYN, SYN-ACK, then the client can send).
- **TLS 1.3 handshake**: 1 round trip.
- **HTTP request/response**: 1 round trip for the client's GET and the server's first bytes.

That's roughly **4 round trips** $\approx$ **320ms** of pure network latency before any HTML, no matter how fast the server or your code. It's why connection reuse (keep-alive, HTTP/2), TLS resumption, QUIC's 0-RTT, and a nearby CDN (chapter 17) matter so much: they remove round trips, and round trips are the thing you can't speed up by optimising code.

CHECKLIST

- ☐ Can you name TLS's three guarantees, and why authentication is the hard one?
- ☐ Do you understand the handshake derives a shared key without sending it, with forward secrecy?
- ☐ Can you explain certificate validation: name, dates, signature chain, trusted root?
- ☐ Do you know what SNI is and why it (without ECH) leaks the hostname?
- ☐ Are your certificates renewed automatically?

GO DEEPER

VISUAL [The Illustrated TLS 1.3 Connection every byte of the handshake](#)

BOOK [High Performance Browser Networking: TLS Ilya Grigorik](#)

REFERENCE [Transport Layer Security MDN](#)

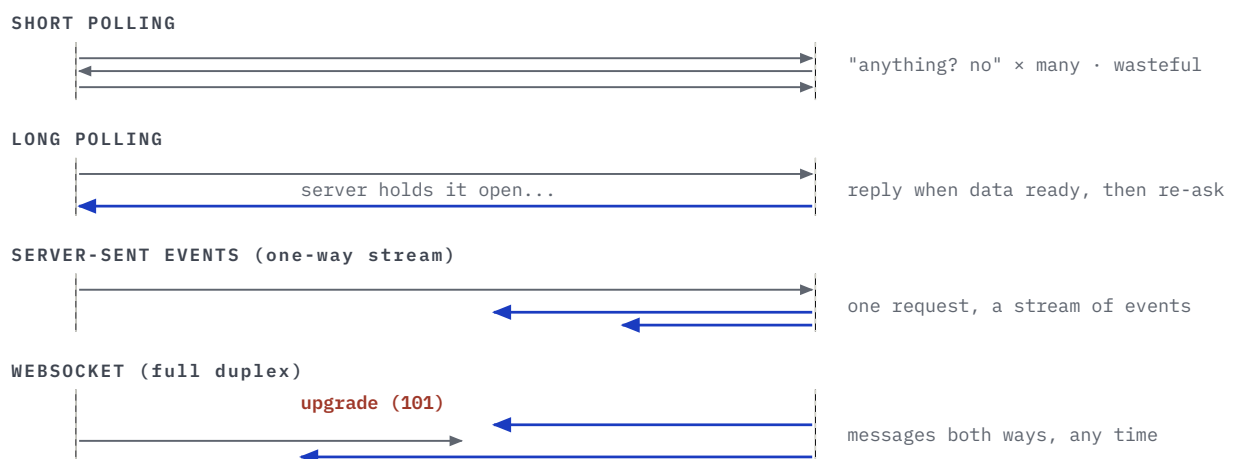
Real-time transports

3 exercises

Plain HTTP is one-shot and client-initiated: you ask, the server answers, done. Real-time features, chat, live scores, collaborative editing, need the server to push, or two sides to talk continuously. There's a ladder of techniques, from faking it with repeated polling up to true bidirectional connections, and the skill is picking the simplest rung that meets the need.

The problem: HTTP only answers when asked

In the request/response model, the server can't speak until the client asks, so "notify me the moment something happens" doesn't fit. Four approaches bridge the gap, each keeping a connection open longer than the last:



Four rungs of the real-time ladder, each holding the connection open longer.

Blue arrows are server-initiated messages, the thing plain request/response can't do. Pick the lowest rung that delivers what you need.

The ladder, rung by rung

- **Short polling.** The client just asks again every few seconds. Trivial to build, works over any HTTP, but wasteful (most replies are "nothing new") and laggy (updates wait for the next poll). Fine for data that changes slowly and tolerates delay.
- **Long polling.** The client asks, and the server *holds the response open* until there's something to send, then answers and the client immediately asks again. Near-real-time with ordinary HTTP, which is why it was the web's workhorse for years, but it's still one message per request and ties up a connection.

- **Server-Sent Events (SSE).** The client opens one long-lived HTTP response and the server *streams* events down it as they happen, a simple text format, one per block. It's built into browsers (`EventSource`), reconnects automatically, and runs over plain HTTP, but it's **one-way** (server to client) and text-only. Ideal for feeds, notifications, live dashboards, progress, and token streaming from an AI model.
- **WebSocket.** A single, persistent, **full-duplex** connection: after an upgrade handshake, either side can send a message at any time, as binary or text, with low overhead. The right tool for chat, multiplayer games, collaborative editing, anything genuinely two-way.

The WebSocket upgrade

A WebSocket is clever about starting: it begins as an ordinary HTTP request that asks to **upgrade** the connection, so it works through the same ports, proxies and firewalls as HTTPS. The server proves it understood by hashing the client's key with a fixed magic string and returning `101 Switching Protocols` ; after that, the TCP connection is a WebSocket and no longer speaks HTTP.

EXAMPLE ws.py · the upgrade handshake (RFC 6455)

```
import base64
import hashlib

# A WebSocket starts life as an ordinary HTTP request that asks to "upgrade" the connection.
# The server proves it understood by hashing the client's key with a fixed magic string.
WS_MAGIC = "258EAF55-E914-47DA-95CA-C5AB0DC85B11" # from RFC 6455

def accept_key(client_key: str) -> str:
    digest = hashlib.sha1((client_key + WS_MAGIC).encode()).digest()
    return base64.b64encode(digest).decode()

def client_upgrade_request(host, path, key):
    return (f"GET {path} HTTP/1.1\r\n"
            f"Host: {host}\r\n"
            f"Upgrade: websocket\r\n"
            f"Connection: Upgrade\r\n"
            f"Sec-WebSocket-Key: {key}\r\n"
            f"Sec-WebSocket-Version: 13\r\n\r\n")

def server_upgrade_response(key):
    return ("HTTP/1.1 101 Switching Protocols\r\n"
            "Upgrade: websocket\r\n"
            "Connection: Upgrade\r\n"
            f"Sec-WebSocket-Accept: {accept_key(key)}\r\n\r\n")

if __name__ == "__main__":
    # The exact example from RFC 6455, section 1.3.
    key = "dGhlIHNhbXBsZSBub25jZQ=="
    print("client key :", key)
    print("accept key :", accept_key(key))
    print("matches RFC:", accept_key(key) == "s3pPLMBiTxaQ9kYGzzhZRbK+x0o=")
    print("\nupgrade handshake over HTTP:")
```

```
print(client_upgrade_request("chat.example", "/ws", key).replace("\r\n",
    "\\r\\n\\n").rstrip())
print(" --- then ---")
print(server_upgrade_response(key).replace("\r\n", "\\r\\n\\n").rstrip())
print("\nafter 101, the TCP connection is a bidirectional WebSocket, no longer HTTP.")
```

The accept-key computation matches the RFC's own test vector exactly, which is how a server (and the browser) confirm the peer really speaks WebSocket and that no cache or proxy mangled the handshake. The magic string serves no cryptographic purpose; it just makes an accidental upgrade impossible.

WebRTC and WebTransport

Two more sit at the specialised ends. **WebRTC** is built for *peer-to-peer* real-time media, video calls, voice, between browsers directly, which means solving hard problems TLS and TCP don't: punching through NATs (chapter 14) with STUN and TURN servers, and tolerating loss over UDP because a dropped video frame should be skipped, not retransmitted (chapter 11). It also offers data channels. It's the right, if heavy, tool for conferencing. **WebTransport** is the newest arrival, reaching cross-browser support in 2026: it runs over HTTP/3 and QUIC, giving multiplexed streams *and* unreliable datagrams over one connection with no head-of-line blocking (chapter 5), effectively a modern WebSocket that also does the lossy-datagram job WebRTC data channels did. For new client-server real-time work on HTTP/3, it's increasingly the answer; WebSocket remains the safe, universal default.

COMMON TRAPS

Reaching for WebSockets when SSE would do. One-way updates don't need a bidirectional connection.

Short polling a fast-changing resource, hammering the server for mostly-empty answers.

Forgetting WebSockets aren't HTTP after the upgrade, so HTTP caching, status codes and many proxies don't apply.

Using WebRTC for simple server push, paying for NAT traversal and media machinery you don't need.

No reconnect/backoff logic: long-lived connections drop, and the client must re-establish gracefully.

EXERCISES

EXERCISE 7.1 Pick the transport

Choose a real-time approach for each, and justify it.

- a. Streaming an AI model's answer token by token to the browser.
- b. A two-player chess game.
- c. A stock ticker updating a few prices every second, display only.
- d. A group video call.

SOLUTION

- a. **Server-Sent Events.** It's one-way (server to client), text, and streaming, exactly SSE's shape, with automatic reconnection for free. No need for a bidirectional WebSocket.
- b. **WebSocket** (or WebTransport on HTTP/3). Moves are bidirectional and need low latency; both players send at any time.
- c. **SSE**, again one-way streaming. Short polling would also work but wastes requests; SSE pushes only when a price changes.
- d. **WebRTC**, which is built for peer-to-peer media with NAT traversal and loss tolerance. The others aren't designed for real-time audio/video.

EXERCISE 7.2 Reason about the handshake

Why does a WebSocket start as an HTTP request instead of just opening a new kind of connection on a new port? And what does the `Sec-WebSocket-Accept` hash protect against?

SOLUTION

Starting as HTTP(S) means WebSockets ride the same ports (80/443), TLS, proxies, firewalls and load balancers as normal web traffic, so they work from locked-down networks where a new port would be blocked, and they can share infrastructure with the rest of the site. The `Sec-WebSocket-Accept` hash (SHA-1 of the client's key plus the magic GUID) proves the server actually understood the WebSocket upgrade and isn't, say, a caching proxy or an old server blindly echoing a `101`; it prevents a non-WebSocket endpoint from accidentally being treated as one, and stops certain cache-poisoning tricks. It's a handshake-confirmation, not encryption, security comes from running over `wss://` (TLS).

EXERCISE 7.3 Design for the drop

Your WebSocket-based collaborative editor works in the office but users on mobile report losing changes when they go through a tunnel or switch networks. What's happening, and how do you make it robust?

SOLUTION

Long-lived connections **drop**, when the network blips, a NAT or load balancer times out an idle connection, or the device changes IP (Wi-Fi to cellular), which breaks the TCP connection the WebSocket rides on (chapter 5, 9). Robustness means treating disconnection as normal: detect it (heartbeat/ping frames with a timeout), reconnect with exponential backoff and jitter, and, crucially, make the application recover state, buffer unsent changes locally and replay them, and on reconnect resynchronise from a server sequence number so nothing is lost or duplicated. For the network-switch case specifically, WebTransport over QUIC (with connection migration, chapter 5) survives the IP change that kills a WebSocket. The deeper lesson: a persistent connection is a convenience, not a guarantee; design as if it will drop, because it will.

CHECKLIST

- ☐ Do you pick the simplest rung, SSE for one-way, WebSocket for two-way, that meets the need?
- ☐ Do you understand SSE is one-way over plain HTTP, WebSocket is full-duplex after an upgrade?
- ☐ Do you know WebRTC is for peer-to-peer media and WebTransport for HTTP/3-era client-server?
- ☐ Does every long-lived connection have heartbeat, reconnect and backoff?
- ☐ Does the application recover cleanly when a connection drops?

GO DEEPER

REFERENCE [WebSockets and Server-Sent Events MDN](#)

SPEC [RFC 6455: The WebSocket Protocol IETF](#)

REFERENCE [WebTransport and WebRTC MDN / webrtc.org](#)

RPC and schemas

3 exercises

A remote procedure call makes talking to another service look like calling a function. Underneath, the arguments are serialised, sent over the network, and decoded on the far side, and the contract that keeps both sides agreeing is a schema. The deep skill is schema evolution: changing that contract without breaking the services already running against it.

Calling a function on another machine

Much of a modern system is services calling other services, and **RPC** (remote procedure call) makes that look like an ordinary function call: you call `getUser(42)` and get a user back, while a framework handles turning the call into bytes, sending them, and turning the reply back into an object. The convenience is real and the danger is that it hides the network: a local call can't be slow for 200 milliseconds, fail halfway, or vanish, and a remote one can (the fallacies of distributed computing, which the distributed-systems stage covers). So treat RPC as a networked operation with timeouts, retries and error handling, not as a function that happens to live elsewhere.

gRPC and the alternatives

The dominant RPC framework, **gRPC**, runs over HTTP/2 (chapter 5), sends compact binary messages, and supports streaming in both directions, not just one request and one response but server streams, client streams and bidirectional streams. It generates strongly-typed client and server code from a schema, so a mismatch is a compile error rather than a runtime surprise. That makes it excellent for *service-to-service* traffic inside a system, where speed and strict contracts matter and humans aren't reading the wire. Its trade-off against plain **REST** (HTTP plus JSON, chapters 3 and 4) is the usual one: REST is human-readable, browser-native and loosely coupled; gRPC is faster, typed and stricter but not directly usable from a browser (which can't control HTTP/2 frames), needing a **gRPC-Web** or Connect proxy. Rule of thumb: REST or JSON for public and browser-facing APIs, gRPC for internal service meshes where you control both ends.

Protocol Buffers: the wire format

gRPC serialises with **Protocol Buffers** (protobuf): you write a schema listing each field with a *number*, and the encoder puts those numbers, not the field names, on the wire, with the values packed tightly. This is both why protobuf is small and fast and why it evolves gracefully. Here is the whole encoding, small enough to hold in your head:

EXAMPLE `protobuf.py` · field numbers are the identity

```

# Protocol Buffers encode a message by FIELD NUMBER, not field name. That one fact is why
# services that deploy independently can evolve their schemas without breaking each other.
def varint(n):
    out = bytearray()
    while True:
        b = n & 0x7F
        n >>= 7
        out.append(b | (0x80 if n else 0))
        if not n:
            return bytes(out)

def read_varint(data, i):
    result = shift = 0
    while True:
        b = data[i]; i += 1
        result |= (b & 0x7F) << shift
        if not (b & 0x80):
            return result, i
        shift += 7

def encode(fields):
    """fields: {field_number: value}. int -> varint (wire type 0), str -> length-delimited
    (2)."""
    out = bytearray()
    for num, value in fields.items():
        if isinstance(value, int):
            out += varint((num << 3) | 0) + varint(value)
        else:
            raw = value.encode()
            out += varint((num << 3) | 2) + varint(len(raw)) + raw
    return bytes(out)

def decode(data, schema):
    """schema: {field_number: (name, type)}. Unknown fields are kept but skipped, forward-
    compat."""
    out, i = {}, 0
    while i < len(data):
        tag, i = read_varint(data, i)
        num, wire = tag >> 3, tag & 0x7
        if wire == 0:
            value, i = read_varint(data, i)
        else:
            length, i = read_varint(data, i)
            value, i = data[i:i + length], i + length
        if num in schema:
            name, typ = schema[num]
            out[name] = value.decode() if typ == "str" else value
        else:
            out[f"<unknown field {num}>"] = value # old code tolerates new fields
    return out

if __name__ == "__main__":
    wire = encode({1: 42, 2: "widget"}) # id=42, name="widget"
    print("on the wire:", wire.hex(), f"({len(wire)} bytes)")

    v1 = {1: ("id", "int"), 2: ("name", "str")}
    print("v1 schema decodes: ", decode(wire, v1))

    # Rename field 2 "name" -> "label": safe, because the wire uses the NUMBER, not the name.
    v2_rename = {1: ("id", "int"), 2: ("label", "str")}
    print("after renaming f2: ", decode(wire, v2_rename))

```

```
# A new field 3 added by a newer sender; the old decoder keeps it as unknown, no crash.
newer = encode({1: 42, 2: "widget", 3: 7})
print("old code, new data: ", decode(newer, v1))

# DANGER: reusing field number 2 for a new meaning silently corrupts old data.
v_bad = {1: ("id", "int"), 2: ("price_cents", "int")}
print("reused number 2:      ", decode(wire, v_bad), "<- price_cents is garbage, not a
      price")
```

The message `{id: 42, name: "widget"}` is ten bytes, and the hex shows why: each field is a tag (the field number and a type) followed by the value, no field names anywhere. JSON would spell out `"id"` and `"name"` on every message; protobuf never does.

Schema evolution: the skill that matters

Services deploy independently, so at any moment an old sender may talk to a new receiver or the reverse, and the schema must change without breaking either. Because the wire identifies fields by *number*, the rules are precise, and the example demonstrates each:

- **Renaming a field is safe.** The example decodes the same bytes with field 2 called `name` and then `label`, both work, because the wire only ever had the number 2. Names are for your code, not the wire.
- **Adding a field is safe,** as long as it gets a *new* number. The old decoder, shown reading a message that includes a new field 3, keeps it as an unknown field and carries on, which is forward compatibility: new senders don't break old receivers.
- **Never change or reuse a field number.** The last line reuses number 2, once a string, now an integer `price_cents`, and old data silently decodes into garbage: `price_cents` comes out as the bytes of "widget". No error, just corruption, which is far worse than a crash. When you remove a field, you mark its number `reserved` so it can never be reused.

These rules, never change or reuse a number, only add with new numbers, are the whole discipline of evolving a binary contract, and they're the same idea as additive, backward-compatible API changes in *The Engineer's Shipyard*. Get them wrong and you get the quietest, nastiest bug there is: two services that both "work" but disagree about what the bytes mean.

Beyond protobuf

Protobuf is one of a family of schema-based binary formats (Avro and Thrift are others), all solving the same problem: a compact, fast, evolvable encoding with a contract both sides share. The alternative, schemaless JSON, trades size and speed for human-readability and needing no shared schema, which is exactly the right trade for public APIs and anything a person debugs by eye. The choice isn't gRPC-versus-REST as tribes; it's binary-and-typed where you own both ends and want speed, text-and-flexible where you don't.

COMMON TRAPS

Treating an RPC like a local call, with no timeout, retry or failure handling.

Reusing or renumbering protobuf fields, silently corrupting data across versions.

Making a breaking schema change and deploying both services at once, hoping nothing is in flight.

Reaching for gRPC in the browser, which can't speak it without a proxy.

Using binary protobuf for a public API that outside developers must read and debug.

EXERCISES

EXERCISE 8.1 Classify the schema changes

For each change to a protobuf message, safe or breaking, and why?

- Rename field 5 from `qty` to `quantity`.
- Add field 9, a new optional `notes` string.
- Change field 3 from `int32` to `string`.
- Delete field 4, then later add field 4 as a new `discount`.

SOLUTION

- Safe.** The wire uses the number 5; the name is only in your code.
- Safe.** A new number means old receivers ignore it and new ones read it, backward and forward compatible.
- Breaking.** Same number, different type: old integers decode as garbage under the new type and vice versa. Add a new field with a new number instead.
- Breaking.** Reusing number 4 means old messages carrying the old field 4 are misread as `discount`. When you delete a field, `reserve` its number forever so this can't happen.

EXERCISE 8.2 REST or gRPC?

Choose for each, and say why.

- a. A public API third-party developers will integrate against.
- b. High-volume calls between your pricing service and your checkout service.
- c. A browser app talking directly to your backend.
- d. Streaming live telemetry from thousands of devices to a collector.

SOLUTION

- a. **REST/JSON.** Public, human-debuggable, browser- and tool-friendly, no schema to distribute, no proxy needed.
- b. **gRPC.** You own both ends; binary protobuf over HTTP/2 is faster and the typed contract catches mismatches at build time.
- c. **REST/JSON**, or **gRPC-Web/Connect** if you want the typed contract, since a browser can't speak raw gRPC.
- d. **gRPC streaming**, whose bidirectional streams and compact binary suit continuous, high-volume telemetry well.

EXERCISE 8.3 Ship the schema change safely

You need to split a protobuf `name` field into `first_name` and `last_name` across two services deployed independently. Outline the safe sequence.

SOLUTION

1. **Add, don't replace:** add `first_name` and `last_name` as new fields with new numbers, keeping the old `name` field.
2. **Deploy the senders** to populate all three fields (old and new) during the transition, so every receiver, old or new, has what it needs.
3. **Deploy the receivers** to read the new fields, falling back to parsing `name` if the new ones are empty (for messages still in flight from not-yet-updated senders).
4. **Once everything reads the new fields** and no old data remains, stop writing `name`, and eventually remove it, `reserve`-ing its number.

At no point is there a flag day where both must change together; each step is backward compatible, which is the whole point of the field-number discipline, and the same expand-then-contract pattern used for database and API migrations elsewhere in the series.

CHECKLIST

- ☐ Do you treat every RPC as a network call, with timeouts and failure handling?
- ☐ Do you understand that protobuf identifies fields by number, not name?
- ☐ Do you only ever add fields with new numbers, and never change or reuse a number?
- ☐ Do you reserve the numbers of removed fields?
- ☐ Do you choose REST/JSON for public and browser APIs, gRPC for internal service traffic?

GO DEEPER

DOCS Protocol Buffers guide protobuf.dev

DOCS What is gRPC? grpc.io

ESSAY Schema evolution in Avro, Protocol Buffers and Thrift [Martin Kleppmann](#)

PART III

Transport

Four chapters down a layer, to how bytes actually get delivered. TCP's reliable connection and its fight against congestion, UDP's deliberate simplicity, and the socket API that every client and server in this book is built on.

TCP: connection and reliability

3 exercises

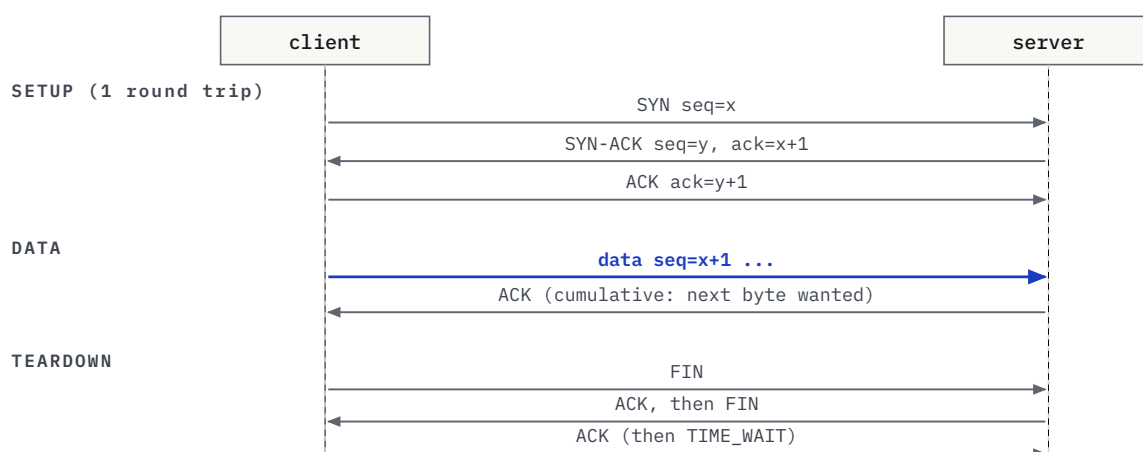
IP delivers packets that may be lost, duplicated or reordered, and makes no promises. TCP builds on top of it the thing almost everything you write depends on: a reliable, ordered stream of bytes between two programs. It does it with a handshake to start, sequence numbers to order, and acknowledgements and retransmission to recover loss.

A reliable stream over an unreliable network

The network layer (IP, chapter 13) is deliberately dumb: it forwards packets and that's all, so a packet can vanish in a full router queue, arrive twice, or overtake an earlier one. TCP's job is to hide all of that and present your program with a clean **byte stream**, what you write comes out the other end, in order, complete, or the connection fails honestly. It achieves this entirely at the two endpoints (the end-to-end principle, chapter 1), using three mechanisms: a handshake, sequence numbers, and acknowledgement with retransmission.

The three-way handshake

Before any data, the two sides synchronise with three messages: the client sends **SYN** (with its starting sequence number), the server replies **SYN-ACK** (acknowledging the client's and sending its own), and the client sends **ACK**. Now both know each other's sequence numbers and that the path works in both directions.



A connection's life: a one-round-trip handshake, a stream of data acknowledged as it arrives, and an orderly four-message teardown. That setup round trip is why connection reuse (chapter 3) and QUIC's faster handshake (chapter 5) matter.

The handshake costs one round trip before a single byte of data, which is the floor on every new TCP connection's latency, and the reason HTTPS, adding TLS's handshake on top, costs two (chapter 6). A connection is identified by the **four-tuple** (source IP, source port, destination IP, destination port), so one server can hold millions of connections at once, each distinguished by the client's address and port, and this is also what NAT (chapter 14) rewrites and what QUIC (chapter 5) escapes to survive network changes.

Sequence numbers and acknowledgement

Every byte in the stream has a **sequence number**, and the receiver sends back **cumulative acknowledgements**: "I have everything up to byte N, send me N next". If a segment is lost, later segments may still arrive, but the ACK stops advancing, pointing at the gap, and the receiver buffers the out-of-order data until the missing piece arrives. The sender keeps a copy of everything un-acknowledged and retransmits it, on a timeout, or faster when it sees repeated ACKs for the same byte (three duplicate ACKs trigger a **fast retransmit**). Here is that whole mechanism:

EXAMPLE tcp.py · reliable, ordered delivery over a lossy channel

```
# TCP turns IP's unreliable, out-of-order packets into a reliable, ordered byte stream, using
# sequence numbers, cumulative acknowledgements, and retransmission. Here is that core.
class Receiver:
    def __init__(self):
        self.expected = 0          # next byte we need, in order
        self.buffer = {}          # out-of-order segments, held until the gap fills
        self.delivered = bytearray()

    def on_segment(self, seq, data):
        """Accept a segment; return the cumulative ACK (the next byte we still want)."""
        if seq <= self.expected < seq + len(data):
            self.buffer[seq] = data
        elif seq > self.expected:
            self.buffer[seq] = data          # arrived early; keep it for later
        while self.expected in self.buffer:  # deliver everything now contiguous
            chunk = self.buffer.pop(self.expected)
            self.delivered += chunk
            self.expected += len(chunk)
        return self.expected                # cumulative ACK

def send_reliably(message, drop_seqs, log):
    """Send 'message' in 4-byte segments over a channel that drops the given seqs ONCE each."""
    rcv = Receiver()
    segments = [(i, message[i:i + 4]) for i in range(0, len(message), 4)]
    acked = 0
    dropped = set(drop_seqs)
    while acked < len(message):
        for seq, data in segments:
            if seq < acked:
                continue                    # already acknowledged, skip
            if seq in dropped:
                dropped.discard(seq)        # the channel drops it this once
                log.append(f"seq {seq}: LOST")
                continue
            ack = rcv.on_segment(seq, data)
            log.append(f"seq {seq} ({data.decode()!r}) -> ACK {ack}")
            acked = ack
```

```

        if acked < seq + len(data):
            break                                # gap: stop, retransmit from here next
            round
    return bytes(rcv.delivered)

if __name__ == "__main__":
    msg = b"HELLO, NETWORK!"
    log = []
    got = send_reliably(msg, drop_seqs={4}, log=log)    # drop the segment starting at byte 4
    print("\n".join(log))
    print("\ndelivered:", got.decode())
    print("correct despite the loss:", got == msg)

```

Watch the ACK in the trace. After the segment at byte 4 is lost, the next segment (byte 8) arrives, but the ACK *stays at 4*, because TCP acknowledges the stream only up to the first gap, that's the cumulative part. The receiver holds byte 8 in its buffer. When the sender retransmits byte 4, the ACK jumps all the way to 12, acknowledging both the retransmitted segment and the buffered one at once. The stream is delivered intact, in order, despite the loss, which is TCP's entire promise kept in a dozen lines.

Teardown, and the states

Closing is a four-message dance: each side sends a **FIN** and gets an **ACK**, because the stream is independently closable in each direction. After its final ACK, a side sits in **TIME_WAIT** for a while before fully releasing the connection, to make sure a late, duplicated packet from this connection can't be mistaken for a new one reusing the same four-tuple. A server with huge numbers of connections in **TIME_WAIT** is a real operational issue you'll meet, and it's a direct consequence of this safety rule. TCP tracks a connection through a set of states (**LISTEN**, **SYN-SENT**, **ESTABLISHED**, **FIN-WAIT**, **TIME_WAIT** and the rest), which the diagnosis tools of chapter 19 let you watch.

COMMON TRAPS

Forgetting the handshake's round trip, and opening a new connection per request when you could reuse one.

.....

Assuming TCP preserves message boundaries. It's a byte *stream*; your two `send` s may arrive as one `recv` or split across several (chapter 12).

.....

Expecting instant failure. A lost connection can take many seconds of retransmission timeouts to be declared dead.

.....

Running out of ephemeral ports or piling up **TIME_WAIT** under high connection churn.

.....

Thinking an ACK means the application processed the data. It means TCP received it; the app may not have read it yet.

.....

EXERCISE 9.1 Read the cumulative ACK

Four 4-byte segments are sent: bytes 0, 4, 8, 12. Segment 4 is lost; 0, 8, 12 arrive. What ACK does the receiver send after each arrival, and why doesn't it jump ahead when 8 and 12 arrive?

SOLUTION

After byte-0 arrives: **ACK 4** (have 0–3, want 4). After byte-8 arrives (4 still missing): **ACK 4** again, the cumulative ACK can only report the stream up to the *first gap*, and there's a hole at 4, so it can't acknowledge 8 even though it has it (it buffers it). After byte-12: **ACK 4** once more. These repeated ACKs for 4 are the **duplicate ACKs** that tell the sender "4 is missing, everything after it is waiting", and three of them trigger a fast retransmit. Once 4 is retransmitted and arrives, the ACK leaps to **16**, acknowledging 4, 8 and 12 all at once from the buffer.

EXERCISE 9.2 Count the round trips, again

A client with 100ms RTT makes a single small HTTPS request to a server it has never connected to (ignore DNS). Break down the round trips from the first packet to the response, and say which ones connection reuse removes.

SOLUTION

- **TCP handshake:** 1 RTT (SYN, SYN-ACK; the client can send data with its ACK).
- **TLS 1.3 handshake:** 1 RTT (chapter 6).
- **HTTP request/response:** 1 RTT.

About 3 RTT = 300ms before the response, all network, none of it the server's code. Keep-alive and HTTP/2 remove the TCP and TLS handshakes on *subsequent* requests (reuse the connection), TLS session resumption removes a TLS round trip on reconnect, and QUIC/HTTP/3 folds transport and TLS into one (and zero on resumption). The whole performance game at this layer is removing round trips, because you cannot make a round trip faster than the speed of light allows (chapter 16).

EXERCISE 9.3 Explain the slow failure

A service calls another over TCP. The callee's machine is yanked off the network (not a clean shutdown). The caller hangs for 30 seconds before erroring. Why so long, and what should the caller have done?

SOLUTION

Because nothing *told* the caller the peer was gone, there was no FIN, just silence. TCP can't distinguish "the network is slow" from "the peer vanished", so it retransmits the un-acknowledged data with exponentially increasing timeouts and only gives up after many attempts, which can be tens of seconds to minutes. This is the fallacy "the network is reliable" made concrete. The caller should not rely on TCP's own timeout: set an **application-level timeout** on the request (a few seconds), so it fails fast and can retry, fall back, or shed load, which is exactly the timeout discipline the distributed-systems and reliability stages insist on. A silent peer is the common case, not the exception.

CHECKLIST

- ☐ Can you describe the three-way handshake and why it costs a round trip?
- ☐ Do you understand cumulative ACKs and how loss is detected and retransmitted?
- ☐ Do you know a connection is identified by its four-tuple?
- ☐ Do you remember TCP is a byte stream with no message boundaries?
- ☐ Do you set application-level timeouts rather than trusting TCP to fail fast?

GO DEEPER

COURSE Stanford CS 144 implement TCP yourself, the lab worth the whole stage

BOOK Computer Networking: A Top-Down Approach Kurose & Ross · the transport-layer chapter

REFERENCE RFC 9293: TCP IETF

TCP: flow and congestion control

3 exercises

A sender has two ways to go too fast: overwhelm the receiver, or overwhelm the network. TCP guards against the first with flow control, a window the receiver advertises, and against the second with congestion control, a window the sender probes blindly, ramping up until loss and backing off. Together they decide how much of a link you actually get.

Two different "slow down" problems

TCP's reliability (chapter 9) is only half the story; the other half is *how fast to send*, and there are two independent limits. **Flow control** stops a fast sender from overwhelming a slow receiver: the receiver advertises, in every ACK, a **receive window**, how much buffer space it has free, and the sender never has more unacknowledged data in flight than that. It's a simple, explicit signal from the one place that knows. **Congestion control** is the hard one: it stops senders from overwhelming the *network* in between, which has no voice to say "stop". The sender must *infer* congestion, and traditionally the only signal it has is packet loss.

The sawtooth: probe, back off, repeat

Lacking a dial marked "network capacity", TCP discovers it by probing. It starts cautiously and ramps up fast (**slow start**: double the window each round trip), until it hits a threshold or loses a packet; then it grows gently (**congestion avoidance**: add one each round trip), nosing upward until loss appears again; and on loss it cuts the window sharply (**multiplicative decrease**: halve it). The result, additive increase and multiplicative decrease (AIMD), is the characteristic sawtooth:

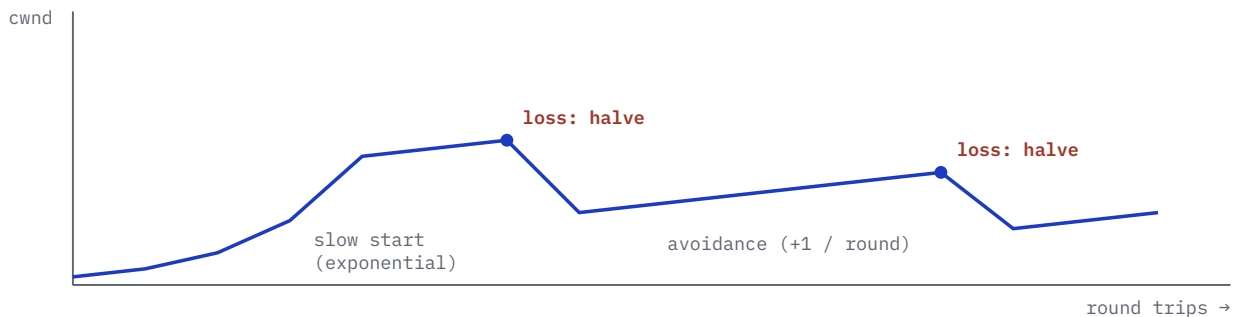
EXAMPLE congestion.py · slow start, avoidance, and the sawtooth

```
# TCP can't be told how fast to send, so it probes: ramp up, and back off when loss appears.
# This is the "AIMD sawtooth" of classic congestion control (Reno-style).
def simulate(rounds, ssthresh=16, loss_rounds=()):
    cwnd = 1                                # congestion window, in segments; grows per round-trip
    history = []
    for r in range(rounds):
        phase = "slow-start" if cwnd < ssthresh else "avoidance"
        history.append((r, cwnd, phase))
        if r in loss_rounds:
            ssthresh = max(1, cwnd // 2)      # multiplicative decrease: halve the target
            cwnd = ssthresh                  # (a timeout would reset cwnd to 1 instead)
        elif cwnd < ssthresh:
            cwnd *= 2                        # slow start: double each round (exponential)
        else:
            cwnd += 1                        # congestion avoidance: additive increase
            (+1/RTT)
    return history
```

```
def bdp(bandwidth_mbps, rtt_ms):
    """Bandwidth-delay product: bytes in flight needed to keep the pipe full."""
    return bandwidth_mbps * 1_000_000 / 8 * (rtt_ms / 1000)

if __name__ == "__main__":
    print("round cwnd phase (one '#' per segment)")
    for r, cwnd, phase in simulate(16, ssthresh=16, loss_rounds={6, 12}):
        bar = "#" * cwnd
        print(f"{r:4} {cwnd:4} {phase:13} {bar}")
    print("\nnote the sawtooth: exponential ramp, then +1/round, halve on loss, repeat.")

    need = bdp(100, 80) # 100 Mbps link, 80ms RTT
    print(f"\nto fill a 100 Mbps link at 80ms RTT you need {need/1024:.0f} KB in flight.")
    print(f"a default 64 KB window would cap you at {64*1024*8 / (80/1000) / 1e6:.0f} Mbps"
          f" -- far below the link. This is why TCP window scaling exists.")
```



TCP keeps probing for more bandwidth and backing off when the network pushes back. The average is your throughput.

The congestion window climbing and collapsing. Loss-based TCP treats a dropped packet as the network saying "too much" and halves its rate, then creeps back up. Many senders doing this share a link roughly fairly.

The code produces exactly this shape: 1, 2, 4, 8, 16 while doubling, then 17, 18 adding one, then a loss halves it to 9, it climbs again, another loss halves it to 7. The *average* of that sawtooth is the throughput you actually get, which is why a lossy link (chapter 5's mobile case) gives you so much less than its raw bandwidth: every loss knocks the window down.

The bandwidth-delay product

How much data must be "in flight" (sent but not yet acknowledged) to keep a link busy? Exactly the **bandwidth-delay product**: bandwidth multiplied by round-trip time. The example computes it: to fill a 100 Mbps link with an 80ms RTT, you need about 977 KB in flight at once. If your window is smaller, you stall waiting for ACKs and never reach the link's speed, the old 64 KB maximum window would cap that fast, far link at a miserable 7 Mbps. This is why high-bandwidth, high-latency paths ("long fat networks", a satellite link, a transcontinental fibre) need large windows, and why **window scaling** was added to TCP. The lesson generalises: on a fast but distant link, throughput is limited by your window and the RTT, not by the bandwidth you're paying for.

CUBIC, BBR, and bufferbloat

Classic loss-based congestion control has a flaw: to detect congestion it must *cause* loss, which means it keeps network buffers full, and a full buffer adds latency to everything behind it. That's **bufferbloat**: a big download makes a video call stutter not because bandwidth ran out but because the buffer filled with the download's packets and everything now waits in a long queue. The default algorithm on most systems, **CUBIC**, is loss-based and prone to this. Google's **BBR** takes a different approach: instead of waiting for loss, it *models* the path, measuring the bottleneck bandwidth and the minimum round-trip time, and sends just enough to fill the pipe without filling the buffers, so it keeps throughput high and latency low. As of 2026 CUBIC is still the common default, BBR is widely used by large services, and the fight against bufferbloat also runs in the routers themselves, through smarter queue management. You rarely choose the algorithm, but knowing why a download tanks your latency, and that it needn't, is the point.

COMMON TRAPS

Expecting to get a link's full bandwidth over a high-latency path with a small window.

Blaming bandwidth for latency under load, when it's bufferbloat, a full queue, not a full pipe.

Treating any loss as a disaster. Loss is how loss-based TCP finds the limit; a little is normal.

Confusing flow control with congestion control. One protects the receiver, the other the network.

Opening many parallel connections to "go faster", grabbing more than a fair share and worsening congestion for everyone.

EXERCISES

EXERCISE 10.1 Size the window

You're copying a large file between two data centres on opposite coasts: a 1 Gbps link with 70ms RTT. The transfer only reaches ~150 Mbps. Using the bandwidth-delay product, explain why, and what to change.

SOLUTION

To fill 1 Gbps at 70ms RTT you need $BDP = 1,000,000,000 / 8 \times 0.070 \approx 8.75$ MB in flight. If the TCP window (limited by the receiver's buffer or OS settings) is, say, ~1 MB, you can only have 1 MB outstanding per RTT, giving $1 \text{ MB} / 0.070\text{s} \approx 114 \text{ Mbps}$, roughly what you see. The fix is to raise the TCP buffer sizes so the window can grow to at least the BDP

(window scaling must be on, which it is by default now, but the buffers cap it). This is the classic "long fat network" problem: on a fast, distant link, your window, not the bandwidth, is the bottleneck.

EXERCISE 10.2 Diagnose the stutter

Whenever someone on the home Wi-Fi starts a big upload, everyone else's video calls stutter and web pages crawl, even though the connection "has plenty of bandwidth". What's happening, and what helps?

SOLUTION

Bufferbloat. The upload's loss-based congestion control fills the router's (often oversized) buffer to find the capacity, and now every other packet, the video call's, the web request's, sits behind a long queue of upload packets, adding hundreds of milliseconds of latency. It's not that bandwidth ran out; it's that the queue got deep. Fixes: enable modern **active queue management** (fq_codel/CAKE) on the router, which keeps queues short and shares them fairly; use a congestion-control algorithm like **BBR** that doesn't fill buffers; and set a sane upload rate limit if the router is dumb. The deep lesson: latency and bandwidth are different resources, and a fat pipe with a bloated buffer can be a slow one.

EXERCISE 10.3 Read the sawtooth

In the chapter's simulation, why does the window grow *exponentially* at first but only *linearly* after the threshold? What would happen to a new connection that stays in slow start the whole time?

SOLUTION

Slow start grows exponentially (doubling) to find the available bandwidth *fast*, since a new connection has no idea how much it can use and starting at one segment per round trip would waste a fast link for ages. But exponential growth would blow straight past the capacity and cause heavy loss, so once TCP is near where it last saw congestion (the threshold), it switches to **additive** increase, nudging up by one per round trip to probe gently. A connection that stayed in slow start would keep doubling until it caused significant loss, overshooting badly, which is exactly why the threshold and the switch to avoidance exist. The two phases are "find the ballpark quickly" then "approach the edge carefully".

CHECKLIST

- ☐ Can you distinguish flow control (protects the receiver) from congestion control (protects the network)?
- ☐ Do you understand slow start, congestion avoidance, and the AIMD sawtooth?
- ☐ Can you compute the bandwidth-delay product and see why a small window throttles a fast, far link?
- ☐ Do you know what bufferbloat is and why a big transfer can wreck latency?
- ☐ Do you know loss-based (CUBIC) and model-based (BBR) control differ, and how?

GO DEEPER

PAPER BBR: Congestion-Based Congestion Control Cardwell et al., ACM Queue

REFERENCE The Bufferbloat project bufferbloat.net

BOOK Computer Networking: A Top-Down Approach Kurose & Ross · congestion control

UDP, and when losing packets is right

3 exercises

UDP is almost nothing: it adds ports and a checksum to raw IP, and that's all, no handshake, no reliability, no ordering, no congestion control. That sounds worse than TCP until you meet the cases where TCP's guarantees are exactly wrong, where a packet that arrives late is useless, and dropping it is better than waiting for it.

The minimal transport

Where TCP is an elaborate machine for reliability and ordering (chapters 9 and 10), UDP is a thin shim over IP. It adds just two things: **ports**, so the packet reaches the right program, and an optional **checksum** to detect corruption. There's no connection to set up, no acknowledgements, no retransmission, no ordering, no flow or congestion control. You send a **datagram** and it either arrives or it doesn't, and you find out neither way. UDP is also **message-oriented**: each `send` is one datagram with preserved boundaries, unlike TCP's shapeless byte stream (chapter 9), which is often reason enough to pick it.

This looks like a downgrade, and for fetching a web page it is, which is why the web runs on TCP. But "the network guarantees nothing and neither do I" is precisely what some applications want, because it hands *them* control over the trade-offs TCP makes for everyone.

When losing a packet is the right call

TCP's cardinal rule, deliver every byte, in order, becomes a liability the moment data has a deadline. Consider a live voice call:

EXAMPLE `udp.py` · why real-time prefers loss to delay

```
# For real-time media, a packet that arrives late is as useless as one that never arrives.
# This shows why UDP (drop and move on) beats TCP (retransmit and wait) for a live stream.
def realtime_playout(n_frames, frame_ms, lost_frame, rtt_ms):
    """Each audio frame has a playout deadline (frame_ms apart). A frame is lost in transit.
    Compare TCP's in-order recovery with UDP's drop-and-conceal."""
    # TCP: in-order delivery, so the lost frame AND every frame behind it wait for the
    # retransmission (~1 RTT later). All of them miss their real-time deadlines.
    retransmit_arrives = lost_frame * frame_ms + rtt_ms
    tcp_late = [f for f in range(n_frames)
                if f >= lost_frame and f * frame_ms < retransmit_arrives]
    # UDP: the lost frame is skipped (a tiny glitch, concealed); every other frame is on time.
    udp_late = [lost_frame]
    return {"tcp_frames_glitched": len(tcp_late), "udp_frames_glitched": len(udp_late)}

if __name__ == "__main__":
    # 20ms audio frames, one lost at frame 10, 120ms round trip.
    r = realtime_playout(n_frames=30, frame_ms=20, lost_frame=10, rtt_ms=120)
```

```
print("one lost frame in a live call (20ms frames, 120ms RTT):")
print(f"  TCP (retransmit + wait): {r['tcp_frames_glitched']} frames arrive too late to
      play")
print(f"  UDP (drop + conceal):    {r['udp_frames_glitched']} frame glitches, the rest on
      time")
print("\nTCP's reliability becomes a liability: it delays good frames to recover a dead
      one.")
print("UDP accepts the one loss and keeps the conversation in sync -- the right trade
      here.")
```

One audio frame is lost. TCP, insisting on in-order delivery, holds back not just the lost frame but *every frame behind it* until the retransmission arrives a round trip later, and in a live call that round trip is an eternity: six frames miss their moment to be played, a long audible gap. UDP simply skips the lost frame, the codec conceals the 20-millisecond hole, and the conversation stays in sync: one tiny glitch instead of a stutter. **A late packet is a useless packet** here, so retransmitting it is worse than forgetting it. That's the whole case for UDP.

Where UDP belongs

- **Real-time media** (voice, video, live streaming): skip losses and conceal them; never stall the stream to recover old data. This is why WebRTC (chapter 7) uses UDP.
- **Live games**: send the *latest* world state; a dropped older state is worthless, so don't retransmit it, send the new one.
- **DNS** (chapter 15): one tiny request, one tiny reply; a connection handshake would cost more than the query, so DNS uses UDP and retries at the application layer if an answer doesn't come.
- **A foundation to build on**: when you want *some* of TCP's guarantees but not all, or want control TCP can't give, you build your own transport over UDP. That's exactly what **QUIC** (chapter 5) is: reliability and ordering *per stream*, its own congestion control, and encryption, all on top of UDP, in user space, free of TCP's rigidity.

What you take on, and the dangers

Choosing UDP means choosing responsibility. Anything you need, reliability, ordering, rate limiting, you must add yourself or deliberately do without. Two dangers in particular. First, **no congestion control** means a naive UDP application can flood a network and contribute to congestion collapse, which is why every serious UDP protocol (including QUIC) implements congestion control of its own, and why you should never just blast datagrams as fast as you can. Second, **amplification attacks**: because UDP has no handshake, an attacker can spoof a victim's address as the source of a small request to a server that replies with a much larger response (DNS and NTP are classic amplifiers), turning the server into an unwitting weapon aimed at the victim. It's a reason UDP services must limit response sizes to unverified senders, and a reason the Mirai-era DDoS attacks hit so hard.

COMMON TRAPS

Reaching for UDP for ordinary data transfer, then reinventing TCP, badly, on top of it.

Sending UDP as fast as you can, with no congestion control, hurting the network and yourself.

Assuming datagrams arrive, in order, intact. None of that is promised; design for loss and reorder.

Building an open UDP service that amplifies small requests into big replies, a DDoS tool.

Forgetting UDP datagrams have a size limit (and fragment badly above the path MTU).

EXERCISES

EXERCISE 11.1 TCP or UDP?

Choose for each, and say why.

- Downloading a software update.
 - A multiplayer shooter sending player positions 60 times a second.
 - A single DNS lookup.
 - Live-streaming a sports match with sub-second latency.
-

SOLUTION

- TCP.** Every byte must arrive, in order; a corrupted update is useless, and there's no deadline that makes a retransmit too late. Reliability is exactly what you want.
- UDP.** Positions are time-sensitive and self-superseding, a dropped old position is worthless because a newer one is already on the way. Retransmitting it would show the player in the past.
- UDP.** One small request and reply; a TCP handshake would triple the cost. The resolver retries at the application layer if needed (and falls back to TCP for large answers).
- UDP-based** (WebRTC or a QUIC-based transport): low latency matters more than perfect delivery, and a dropped frame is concealed rather than stalling the stream. A buffered, non-live stream, by contrast, can use TCP.

EXERCISE 11.2 Reason about the deadline

In the chapter's model, why does one lost frame glitch *six* frames under TCP but only one under UDP? What would reduce the TCP number (short of switching to UDP)?

SOLUTION

Under TCP, the lost frame can't be delivered until its retransmission arrives ~1 RTT (120ms) later, and because TCP delivers *in order*, every frame that arrives in the meantime is held in the buffer behind the gap. With 20ms frames, 120ms of waiting means six frames (the lost one plus the five that arrive during the wait) all miss their playout deadlines, this is head-of-line blocking (chapter 5) applied to audio. UDP delivers each frame independently, so only the actually-lost one glitches. To reduce the TCP number without leaving TCP you'd need a bigger **jitter/playout buffer** (delay playback enough to cover a retransmit), but that adds constant latency to the whole call, which for a conversation is itself a problem. There's no free lunch: TCP trades latency for completeness, and real-time wants the opposite trade.

EXERCISE 11.3 Build on UDP responsibly

You're writing a custom protocol over UDP for a game. List what you must add yourself, and one danger to avoid.

SOLUTION

- **Only the reliability you actually need:** perhaps reliable, ordered delivery for chat and inventory changes, but unreliable, unordered for position updates (send the latest, drop the rest). Mixing both per message type is the point of building your own.
- **Congestion control**, or at least rate limiting, so you don't flood the network, this is non-negotiable and the reason QUIC includes it.
- **Sequence numbers** to detect loss and reordering where you care, and to discard stale data.
- **Message framing within the datagram limit**, keeping messages under the path MTU to avoid fragmentation.
- **The danger:** an unauthenticated, amplifying endpoint (small request, big reply) that attackers can abuse for DDoS, require a handshake or a token before sending large responses to a new address, as QUIC does. Honestly, unless you have a strong reason, **building on QUIC** rather than raw UDP gives you most of this correctly for free.

CHECKLIST

- ☐ Can you explain what UDP adds to IP, and what it deliberately leaves out?
- ☐ Do you understand why a late packet is useless for real-time, so loss beats delay?
- ☐ Do you know UDP is message-oriented, where TCP is a byte stream?
- ☐ If you build on UDP, do you add congestion control and only the reliability you need?
- ☐ Do you understand UDP amplification and why open UDP services are a DDoS risk?

GO DEEPER

BOOK [Computer Networking: A Top-Down Approach](#) Kurose & Ross · UDP and principles of reliable transfer

REFERENCE [UDP MDN](#)

SPEC [RFC 9000: QUIC](#) UDP done right, for reference

Sockets: the API underneath everything

3 exercises

A socket is the operating system's handle for a network connection, and it's the API every client and server you have ever used is built on, your browser, curl, a database driver, a web framework. Learn the handful of calls and the one gotcha that TCP is a stream of bytes, not messages, and you can build the servers the rest of the web is made of.

The API every program shares

Underneath every HTTP library and web framework is the **Berkeley socket API**, a small set of system calls, unchanged for decades, that give a program a file-like handle onto a TCP or UDP connection. A **server** creates a socket, **binds** it to an address and port, **listens**, and **accepts** connections, getting a fresh socket for each one; a **client** creates a socket and **connects** to an address; both then **send** and **recv** bytes and **close** when done. That's the whole vocabulary, and this stage's deliverable, an HTTP server on raw sockets, is exactly these calls plus the HTTP parsing of chapter 3. Here is a real, working pair:

EXAMPLE sockets.py · a real TCP client and server

```
import socket
import struct
import threading

# A socket is the OS handle for a network connection. Every client and server you have used,
# browsers, curl, databases, is built on these calls. Here is a real, working pair.
def recv_exact(conn, n):
    """TCP is a byte STREAM, not messages: recv() may return fewer bytes than asked.
    To read a whole message you must loop. This is the single most common socket bug."""
    buf = b""
    while len(buf) < n:
        chunk = conn.recv(n - len(buf))
        if not chunk:
            raise ConnectionError("peer closed mid-message")
        buf += chunk
    return buf

def recv_message(conn):
    """Frame messages with a 4-byte length prefix, since TCP won't do it for us (chapter 9)."""
    (length,) = struct.unpack(">I", recv_exact(conn, 4))
    return recv_exact(conn, length)

def send_message(conn, data):
    conn.sendall(struct.pack(">I", len(data)) + data)

def server(ready, host="127.0.0.1"):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((host, 0))
```

a TCP socket

0 = pick a free port

```

s.listen()                                # become a listening server
ready.append(s.getsockname())             # tell the client our port
conn, peer = s.accept()                   # blocks until a client
    connects
with conn:
    msg = recv_message(conn)
    send_message(conn, b"echo: " + msg)    # echo it back, framed
s.close()

if __name__ == "__main__":
    ready = []
    t = threading.Thread(target=server, args=(ready,))
    t.start()
    while not ready:
        pass
    host, port = ready[0]
    print(f"server listening on {host}:{port}")

    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client.connect((host, port))           # the TCP handshake happens
        here
    print("connected; local 4-tuple:", client.getsockname(), "->", client.getpeername())
    send_message(client, b"hello sockets")
    reply = recv_message(client)
    client.close()
    t.join()
    print("server replied:", reply.decode())
    print("round trip over a real socket:", reply == b"echo: hello sockets")

```

It genuinely runs: the server binds to a free port, the client connects (the TCP handshake of chapter 9 happens inside `connect`), sends a message, and gets it echoed back. Notice the **four-tuple** printed, `(127.0.0.1, 49172) -> (127.0.0.1, 49171)`, the source and destination addresses and ports that uniquely identify this connection, which is why one listening server can hold thousands of simultaneous connections on a single port: each is a different four-tuple, and each gets its own socket from `accept`.

The gotcha: it's a stream, not messages

The single most common socket bug follows directly from chapter 9: **TCP is a byte stream with no message boundaries**. Your one `send` of 1000 bytes may arrive as one `recv` of 1000, or two of 500, or 1000 of one, and two `send` s may arrive merged into a single `recv`. The network decides, based on segmentation and timing, not on how you called `send`. So you cannot assume `recv` returns a whole message; you must **frame** the data yourself, and loop until you have all of it. The example does this with a 4-byte length prefix and a `recv_exact` loop, and HTTP (chapter 3) does it with `Content-Length` or chunked encoding. Code that does `data = conn.recv(4096)` and parses `data` as a complete message works on your laptop and fails in production the first time a message is split across packets, which is a rite of passage every socket programmer goes through once.

Handling many connections

The example serves one connection and exits. A real server must handle many at once, which is where the deliverable's "concurrent connections" requirement bites, and there are three classic approaches:

- **A thread (or process) per connection.** Simple to write: `accept` in a loop, hand each new socket to a worker. It's fine for hundreds of connections but each thread costs memory and the OS struggles past some thousands, the historical **C10k problem** (ten thousand connections).
- **An event loop (async I/O).** One thread watches many sockets at once with `select`, `epoll` or `kqueue`, doing a little work on whichever is ready, never blocking on any one. This is how nginx, Node.js and Python's `asyncio` handle tens of thousands of connections cheaply, and it's why the networking world went event-driven.
- **A hybrid:** a pool of threads, each running an event loop, to use every CPU core, which is what most high-performance servers actually do.

The key realisation is that a networked server spends most of its time *waiting*, for the next byte, for the client to read, so the scarce resource isn't CPU but the ability to wait on many things at once without a thread tied up per wait. That's the whole motivation for non-blocking I/O.

UDP sockets, and the shape of the two

A UDP socket (chapter 11) uses the same API with a different type (`SOCK_DGRAM` instead of `SOCK_STREAM`) and a simpler shape: no `listen`, no `accept`, no connection at all, you just `sendto` an address and `recvfrom` whoever sends you a datagram. Each `recvfrom` returns exactly one datagram with its boundary intact, the message-oriented nature from chapter 11, so the stream-framing gotcha doesn't apply, but loss and reordering do. The two socket types are the API's reflection of the two transports: a stream you must frame, or datagrams you must make reliable.

COMMON TRAPS

Treating one `recv` as one message. Frame your data and loop until you have it all.

Forgetting to loop `send` too (or use `sendall`): `send` may write fewer bytes than given.

A thread per connection at large scale, hitting the C10k wall instead of using an event loop.

Blocking the whole server on one slow client, rather than handling connections concurrently.

Leaking sockets by not closing them, running out of file descriptors.

EXERCISE 12.1 Find the framing bug

A colleague's server does `line = conn.recv(1024).decode()` and splits it on newlines to get commands. It works in testing and corrupts commands in production. What's wrong, and how do you fix it?

SOLUTION

It assumes one `recv` returns exactly the bytes of one logical chunk, but TCP is a stream: a command may be split across two `recv` calls (so the first ends mid-command) or several commands may arrive in one `recv` (and the last may be incomplete). In testing, messages are small and fast, so they usually arrive whole; in production, under real timing and larger payloads, they don't. The fix is to **buffer** across `recv` calls and only process complete messages: accumulate incoming bytes, split off whole newline-terminated commands, and keep any trailing partial command in the buffer for the next read. That's the same job as the chapter's length-prefix framing, done with a delimiter instead.

EXERCISE 12.2 Scale the server

A thread-per-connection server handles 500 users fine but falls over at 20,000. What's happening, and what are the options?

SOLUTION

Each connection has its own OS thread, and 20,000 threads cost gigabytes of stack memory and swamp the scheduler with context switches, the **C10k problem**. Most of those threads are just *waiting* on idle sockets, so the memory is pure waste. Options: switch to an **event loop** (async I/O with `epoll` / `kqueue`), where one thread services whichever of the 20,000 sockets has data ready, no thread-per-connection cost; or use a small pool of event-loop threads to use all cores. This is exactly why nginx and Node.js are event-driven, and why Python has `asyncio`. The deliverable's "concurrent connections" requirement is really an invitation to meet this problem.

EXERCISE 12.3 Trace the calls

List the socket calls, in order, that a browser and a web server make to exchange one request, from the server starting up to the connection closing.

SOLUTION

Server, at startup: `socket()` → `bind((host, 443))` → `listen()` , then loops on `accept()` , which blocks until a client arrives. **Browser:** `socket()` → `connect((server_ip, 443))` , which performs the TCP handshake (chapter 9). **Server:** `accept()` returns a new socket for this connection. **Then:** the browser `send()` s the HTTP request; the server `recv()` s it (looping until the full request is read, per framing); the server `send()` s the response; the browser `recv()` s it (again looping). **Finally:** both `close()` (or keep the socket for the next request, chapter 3's keep-alive). The listening socket stays open for the next client. Every HTTP library you use is doing exactly this underneath.

CHECKLIST

- ☐ Do you know the server calls (`socket`, `bind`, `listen`, `accept`) and client calls (`socket`, `connect`)?
- ☐ Do you frame messages and loop `recv` , never assuming one read is one message?
- ☐ Do you understand the four-tuple and how one port serves many connections?
- ☐ Do you know when to use threads versus an event loop for concurrency?
- ☐ Could you build the deliverable: an HTTP/1.1 server on raw sockets?

GO DEEPER

GUIDE [Beej's Guide to Network Programming](#) the classic, free socket tutorial

REFERENCE [Socket programming HOWTO](#) Python docs

ESSAY [The C10K problem](#) Dan Kegel

PART IV

Addressing and naming

Three chapters on how packets find their destination. The IP addresses that identify every host and the subnets that group them, how routers and NAT move packets between networks, and DNS, the directory that turns a human name into an address, and the thing that breaks most often.

IP: addressing and subnets

3 exercises

Every host on the internet has an IP address, and an address plus a prefix length splits into a network part and a host part. That split is the whole basis of routing: routers don't track individual hosts, they track networks, and forward a packet toward the network its destination address falls in. Learn to read CIDR notation and you can read the internet's map.

Addresses, and the two versions

IP's one job is to give every host an address and deliver packets to it. There are two versions in use. **IPv4** uses 32-bit addresses, written as four bytes (`192.168.1.50`), which gives about 4.3 billion, long since exhausted, which is why we have NAT (chapter 14) and IPv6. **IPv6** uses 128-bit addresses, written in hex groups (`2001:db8:abcd:12::1`), a space so vast (3.4×10^{38}) that every device can have a real, globally routable address with no sharing. IPv6 passed 50% of Google's traffic in 2026 and keeps climbing, so you need to be fluent in both; the concepts are the same, only the size differs.

CIDR: the network/host split

An address alone doesn't say where the network ends and the host begins; a **prefix length** does. `192.168.1.0/24` means "the first 24 bits are the network, the last 8 identify hosts within it", which is **CIDR** notation, and it's the language of every routing table and firewall rule. The arithmetic is worth seeing concretely:

EXAMPLE `ip.py` · CIDR, subnets, and the membership test

```
import ipaddress

# An IP address plus a prefix length (CIDR) splits into a network part and a host part.
# The /N means "the first N bits identify the network"; the rest identify the host.
def describe(cidr):
    net = ipaddress.ip_network(cidr, strict=False)
    hosts = list(net.hosts())
    return {
        "network": str(net.network_address),
        "prefix": net.prefixlen,
        "total_addresses": net.num_addresses,
        "usable_hosts": len(hosts),
        "first_host": str(hosts[0]) if hosts else None,
        "last_host": str(hosts[-1]) if hosts else None,
        "broadcast": str(net.broadcast_address) if net.version == 4 else "(none in IPv6)",
    }

def in_network(ip, cidr):
    return ipaddress.ip_address(ip) in ipaddress.ip_network(cidr)
```

```

if __name__ == "__main__":
    for k, v in describe("192.168.1.0/24").items():
        print(f" {k:16}: {v}")

    print("\nsplitting a /24 into four /26 subnets (e.g. one per team):")
    for sub in ipaddress.ip_network("192.168.1.0/24").subnets(new_prefix=26):
        print(f" {sub} ({sub.num_addresses - 2} usable hosts)")

    print("\nrouting is a membership test (longest-prefix match):")
    for ip in ["192.168.1.50", "192.168.2.50", "10.0.0.1"]:
        print(f" {ip:14} in 192.168.1.0/24? {in_network(ip, '192.168.1.0/24')}")

    print("\nspecial ranges you must recognise:")
    for ip in ["10.0.0.5", "127.0.0.1", "169.254.169.254", "8.8.8.8"]:
        a = ipaddress.ip_address(ip)
        kind = ("private" if a.is_private and not a.is_loopback and not a.is_link_local
                else "loopback" if a.is_loopback else "link-local" if a.is_link_local
                else "global")
        print(f" {ip:16} {kind}")

    v6 = ipaddress.ip_network("2001:db8:abcd:12::/64")
    print(f"\nIPv6 {v6}: {v6.num_addresses:.3g} addresses in one standard /64 subnet")

```

A `/24` holds 256 addresses, of which 254 are usable hosts (the first is the network address, the last is the broadcast address in IPv4). Shrinking the prefix grows the network; growing it shrinks it, a `/26` has 64 addresses, a `/16` has 65,536. **Subnetting** is splitting one network into smaller ones, as the example splits a `/24` into four `/26` s, which is how an organisation carves its address space into per-team, per-service or per-availability-zone pieces.

Routing is a membership test

Here's why the split matters: a router doesn't store a route for every one of the billions of hosts; it stores routes for *networks*, and to forward a packet it finds which network the destination address belongs to. The example's `in_network` check is that decision in miniature. Real routers do **longest-prefix match**: a packet for `192.168.1.50` might match both `192.168.0.0/16` and `192.168.1.0/24`, and the router picks the *more specific* one (the longer prefix), because it's the more precise route. This aggregation, millions of hosts summarised as a handful of network prefixes, is what keeps the internet's routing tables merely enormous rather than impossible, and it's the mechanism chapter 14 builds on.

Special addresses to recognise

Not every address is a public destination, and confusing the categories causes real bugs and real breaches. The ranges to know: **private** addresses (`10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`) are reusable inside any network and never routed on the public internet, which is what NAT exists to bridge; **loopback** (`127.0.0.1`) is the host talking to itself; and **link-local** (`169.254.0.0/16`) is for one network segment only, including the cloud metadata

address `169.254.169.254` whose role in the Capital One breach the security stage covers. The example classifies each, and that classification is exactly what a server-side-request-forgery guard checks before fetching a URL (*The Engineer's Citadel*, chapter 6).

Why IPv6, and why it took so long

IPv4 ran out of addresses more than a decade ago; IPv6 is the designed-in fix, with room for every device to have its own global address and no need for the address-sharing contortions of NAT. Its slow adoption, 18 years to reach half of traffic, is a case study in network effects: there was no single day to switch, every participant had to move independently, and NAT worked well enough to remove the urgency. The practical upshot for you: write code and infrastructure that handles both, parse and store addresses with a library that understands IPv6 (as the example does), never assume an address is four dotted numbers, and know that on an IPv6 network there's often no NAT, so every host is directly addressable, which changes how you think about firewalls.

COMMON TRAPS

Parsing IP addresses by hand (splitting on dots), which breaks on IPv6 and edge cases. Use a library.

Off-by-one on subnet sizes: a `/24` is 256 addresses but 254 usable hosts.

Confusing private and public addresses, or assuming a private IP can be reached from outside.

Hardcoding IPv4 assumptions, so your software, logs or allowlists break on IPv6.

Trusting a source IP for identity, when NAT and proxies mean many users share one, and addresses are easily spoofed on UDP.

EXERCISES

EXERCISE 13.1 Do the subnet maths

You're given `10.20.0.0/16` for a data centre and need subnets of at least 500 hosts each, one per rack.

- What prefix length gives at least 500 usable hosts per subnet?
- How many such subnets fit in the `/16`?
- Is `10.20.3.17` in the first such subnet?

SOLUTION

- a. A `/23` has 512 addresses, 510 usable, which clears 500; a `/24` (254 usable) wouldn't. So `/23`.
- b. From a `/16` to `/23` is 7 extra bits, so $2^7 = 128$ subnets.
- c. The first `/23` is `10.20.0.0/23`, covering `10.20.0.0` to `10.20.1.255`. `10.20.3.17` is in the *second* `/23` (`10.20.2.0/23`), so **no**. Running the chapter's `in_network("10.20.3.17", "10.20.0.0/23")` confirms it.

EXERCISE 13.2 Longest-prefix match

A router's table has routes for `0.0.0.0/0` (the default), `10.0.0.0/8`, and `10.1.2.0/24`. Which route is used for a packet to `10.1.2.5`? To `10.9.9.9`? To `8.8.8.8`?

SOLUTION

- `10.1.2.5` matches all three; the router picks the **longest prefix**, `10.1.2.0/24`, the most specific route.
- `10.9.9.9` matches `10.0.0.0/8` and the default, but not the `/24`; longest match is `10.0.0.0/8`.
- `8.8.8.8` matches only `0.0.0.0/0`, the **default route**, the catch-all for "anything I don't have a more specific route for", which on most machines points at the gateway toward the internet.

Longest-prefix match is why you can have a broad default route and override it for specific networks, the foundation of how routing tables express policy.

EXERCISE 13.3 Classify for an SSRF guard

A feature fetches a URL a user supplies. Before connecting, which address ranges must it refuse, and why? (This is the networking half of the chapter-6 security guard in the security stage.)

SOLUTION

Resolve the hostname to an address and refuse any that is not a **global** (public) address: block **private** ranges (`10/8`, `172.16/12`, `192.168/16`) so the fetch can't reach internal services; **loopback** (`127/8`, and IPv6 `::1`) so it can't hit services on the host itself; **link-local** (`169.254/16`) above all, because that's the cloud metadata endpoint that hands out credentials; and other reserved ranges (multicast, `0.0.0.0`, IPv6 unique-local `fc00::/7`). The chapter's

classification, using `is_private`, `is_loopback`, `is_link_local`, is exactly the check, and it must run on the *resolved* address, re-checking after any redirect, since a public hostname can resolve to a private IP. This is where networking knowledge becomes security.

CHECKLIST

- ☐ Can you read CIDR notation and compute a network's size and host range?
- ☐ Can you subnet a network into smaller ones for a given host count?
- ☐ Do you understand routing as longest-prefix membership matching?
- ☐ Do you recognise private, loopback and link-local ranges on sight?
- ☐ Does your code handle IPv6, not just IPv4?

GO DEEPER

REFERENCE [The ipaddress module Python docs](#)

BOOK [Computer Networking: A Top-Down Approach Kurose & Ross · the network-layer chapter](#)

REFERENCE [Understanding IP addressing RIPE NCC](#)

Routing and NAT

3 exercises

The internet is a network of networks, and routing is how a packet crosses between them, each router making a local decision toward the destination, with no one in charge of the whole path. NAT is the hack that let a world of private networks share a dwindling supply of public addresses, and both routing's trust-based fragility and NAT's broken end-to-end addressing shape the networks you build on.

Routing: local decisions, global delivery

No router knows the whole path to a destination; each one just knows the *next hop*. A packet for a distant server is handed from your machine to your gateway, to your ISP, across the internet's backbone, router by router, each consulting its routing table, matching the destination against its known networks by longest prefix (chapter 13), and forwarding one hop closer. The destination is reached not by anyone planning the route but by a chain of local decisions, which is what makes the internet both scalable (no central map) and resilient (a packet routes around a failed link). Your own machine's table is tiny, usually just "my local network is here, everything else goes to the default gateway", and the complexity lives in the core.

Autonomous systems and BGP

The internet is divided into about a hundred thousand **autonomous systems** (AS), independently operated networks, ISPs, cloud providers, large companies, each with a number. Between them, routes are exchanged by the **Border Gateway Protocol** (BGP): each AS announces "I can reach these network prefixes", and these announcements propagate so every AS learns a path to every prefix. BGP is the internet's control plane, and its defining feature is that it runs on *trust*: historically an AS could announce a prefix it didn't own and others would believe it, which is how **BGP hijacks** (accidental or malicious) have rerouted traffic for whole countries. It's also how the internet's most dramatic outages happen: in 2021 a configuration error made Facebook *withdraw* the BGP routes to its own DNS servers, and in minutes Facebook, Instagram and WhatsApp vanished from the internet entirely, not because the servers were down but because no one could find a route to them. Routing is the layer that decides whether you exist on the internet at all, and it's secured by convention more than by cryptography (though RPKI is slowly changing that).

NAT: sharing a scarce address

IPv4 ran out of addresses (chapter 13), so almost no home or office device has a public one. Instead they sit on private addresses behind a router doing **Network Address Translation**: on the way out, the router rewrites the packet's private source address and port to its own

single public address and a chosen port, remembering the mapping; on the way back, it reverses the rewrite and delivers the reply to the right inside device. One public address serves a whole household or building. Here is the mechanism:

EXAMPLE nat.py · many private hosts, one public IP

```
# NAT lets many devices on private addresses share one public IP. The NAT box rewrites the
# source address and port on the way out, and reverses the rewrite on the replies coming back.
class NAT:
    def __init__(self, public_ip):
        self.public_ip = public_ip
        self.next_port = 40000
        self.outbound = {} # (inside_ip, inside_port, dst) -> public_port
        self.inbound = {} # public_port -> (inside_ip, inside_port)

    def translate_outbound(self, inside_ip, inside_port, dst):
        key = (inside_ip, inside_port, dst)
        if key not in self.outbound:
            port = self.next_port
            self.next_port += 1
            self.outbound[key] = port
            self.inbound[port] = (inside_ip, inside_port)
        port = self.outbound[key]
        return (self.public_ip, port) # what the internet sees as the source

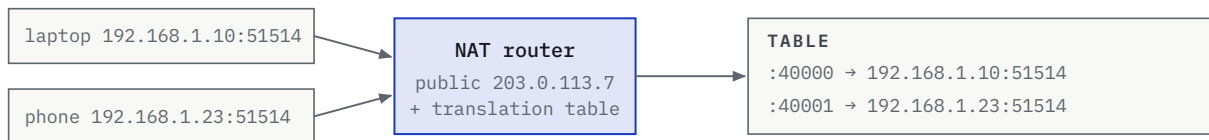
    def translate_inbound(self, public_port):
        """A reply arrives for public_ip:public_port; map it back to the real inside host."""
        return self.inbound.get(public_port) # None = unsolicited; NAT drops it

if __name__ == "__main__":
    nat = NAT("203.0.113.7") # the home router's single public IP
    dst = ("93.184.216.34", 443) # some web server

    # Two different devices behind the router open connections to the same server.
    laptop = nat.translate_outbound("192.168.1.10", 51514, dst)
    phone = nat.translate_outbound("192.168.1.23", 51514, dst) # same inside port, by chance
    print("laptop 192.168.1.10:51514 ->", f"{laptop[0]}:{laptop[1]}")
    print("phone 192.168.1.23:51514 ->", f"{phone[0]}:{phone[1]}")
    print("both appear to the internet as one IP:", laptop[0] == phone[0] == "203.0.113.7")

    # Replies come back addressed to the public IP + the mapped port; NAT sends each home.
    print("\nreply to port", laptop[1], "-> inside", nat.translate_inbound(laptop[1]))
    print("reply to port", phone[1], "-> inside", nat.translate_inbound(phone[1]))
    print("unsolicited packet to port 55555 ->", nat.translate_inbound(55555), "(dropped)")
```

PRIVATE NETWORK



To the internet both are 203.0.113.7, on different ports. Replies come back by port and are sent home.

But an unsolicited inbound packet matches no table entry, so NAT drops it:
incoming connections don't work without port forwarding or hole-punching (STUN/TURN).

NAT multiplexes many private hosts onto one public address by remembering a port mapping per connection. It works brilliantly for outbound traffic and breaks inbound: nothing outside can start a connection to a device that isn't in the table.

What NAT breaks

NAT quietly solved address exhaustion and quietly broke the internet's original promise that any host can talk to any host. Because the mapping is created by *outbound* traffic, an *inbound* connection to a device behind NAT has nowhere to go, the router has no table entry for it, so it's dropped (the example's last line). That's why running a server at home needs **port forwarding**, why peer-to-peer apps like video calls need **hole-punching** through STUN and relaying through TURN servers (the machinery behind WebRTC, chapter 7), and why "just connect directly to the other user" is so much harder than it sounds. At the ISP scale, **carrier-grade NAT (CGNAT)** shares one public address among many *customers*, compounding the problem and meaning a source IP no longer identifies even a household. IPv6, with an address for every device, removes the need for NAT entirely, which is one of the strongest arguments for finishing the migration.

COMMON TRAPS

Assuming you can connect *to* a device that's behind NAT without forwarding or hole-punching.

.....
Trusting a source IP to identify a user, when NAT and CGNAT put many users behind one.

.....
Forgetting BGP and DNS are the control plane: a routing or DNS mistake removes you from the internet, servers fine or not.

.....
Treating routing as secure. BGP runs on trust; hijacks and leaks happen.

.....
Designing peer-to-peer features without planning for NAT traversal.

EXERCISES

EXERCISE 14.1 Trace the translation

Using the chapter's NAT, your laptop (`192.168.1.10:51514`) and your flatmate's laptop (`192.168.1.99:51514`) both load the same website. What does the website see as the source of each? When its replies come back, how does the router know which laptop each is for?

SOLUTION

The website sees both as coming from the router's single public IP (`203.0.113.7`), but on *different public ports* (say 40000 and 40001), because the NAT assigns a distinct public port per connection even though both laptops happened to use the same private port 51514. When a reply arrives addressed to `203.0.113.7:40000` , the router looks up port 40000 in its table, finds `192.168.1.10:51514` , rewrites the destination, and delivers it to your laptop; port 40001 goes to your flatmate's. The **port** is the demultiplexing key. This is also why, to an outside observer, everyone in a household (or behind CGNAT, a whole neighbourhood) shares one address, so IP-based rate limits or blocks catch innocent neighbours.

EXERCISE 14.2 The disappearing company

A company pushes a routing change and within minutes its website, API and even its internal badge readers are unreachable from everywhere, though the servers are running fine. What class of failure is this, and what's the lesson?

SOLUTION

This is the Facebook-2021 pattern: a BGP (and/or DNS) change removed the routes that tell the rest of the internet how to reach the company's networks, so the servers are up but **unreachable, there's no path to them**. It's a control-plane failure, not a server failure, which is why "the servers are fine" is both true and useless. The lessons: BGP and DNS are load-bearing infrastructure whose mistakes have no blast-radius limit, so changes to them need the same care, staging and rollback as any deploy; recovery can require *physical* access if the same outage locked out the remote-management and badge systems (as it did for Facebook); and you should monitor reachability from *outside* your network, because from inside, everything looks fine.

EXERCISE 14.3 Connect two peers behind NAT

Two users, each behind their home NAT, want a direct video call. Neither can accept an inbound connection. How is this solved in practice?

SOLUTION

- **A rendezvous (signalling) server** that both peers *can* reach (it has a public address) introduces them and exchanges their connection info.
- **STUN**: each peer asks a STUN server "what public address and port does my NAT show for me?", discovering its external mapping so the other side knows where to aim.
- **Hole-punching**: both peers send packets toward each other's discovered public address at the same time, so each one's NAT creates an outbound mapping that then accepts the other's incoming packets, tricking two NATs into letting a "direct" connection form.
- **TURN** as a fallback: when hole-punching fails (strict NATs), traffic is relayed through a public TURN server, slower and costlier, but it always works.

This whole dance, standardised as ICE, is why WebRTC (chapter 7) is heavy, and it exists entirely because NAT broke the ability to simply connect to a host. On IPv6 with public addresses, much of it is unnecessary.

CHECKLIST

- ☐ Do you understand routing as a chain of local next-hop decisions, not a planned path?
- ☐ Do you know what an autonomous system and BGP are, and why routing is trust-based?
- ☐ Can you explain how NAT lets many hosts share one public IP, and what it breaks?
- ☐ Do you know why inbound connections to NATed hosts need forwarding or hole-punching?
- ☐ Do you avoid trusting a source IP to identify a user?

GO DEEPER

REFERENCE [What is BGP? Cloudflare Learning](#)

REPORT

More details about the October 4 outage Meta Engineering, 2021

BOOK

Computer Networking: A Top-Down Approach Kurose & Ross · routing and the network core

DNS: the internet's directory

3 exercises

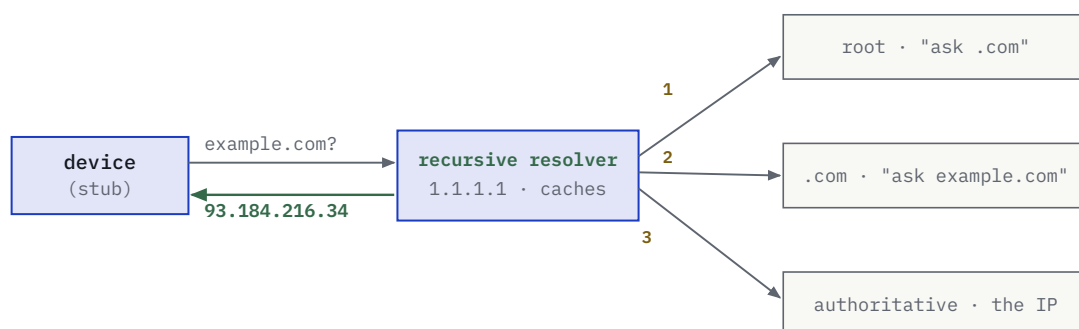
DNS turns a name a human can remember into an address a packet can reach, by walking a global hierarchy and caching the answer for its TTL. It runs before almost every connection you make, it's the layer most of your "mysterious" failures hide in, and the oldest joke in operations, "it's always DNS", is a joke because it's usually true.

Names to addresses

Packets travel to IP addresses (chapter 13), but nobody types `93.184.216.34`; they type `example.com`. The **Domain Name System** is the distributed directory that translates between them, and it runs before nearly every connection, so its speed and reliability underpin everything above it. It's also a system you'll configure often, pointing a domain at a server, a CDN, a mail provider, so understanding its hierarchy and its caching is daily-useful, not just theory.

The hierarchy, and who does the walking

No single server knows every name; the namespace is a tree, resolved from the top. A name like `example.com` is found by asking the root servers "who handles `.com`?", then the `.com` servers "who handles `example.com`?", then `example.com`'s own **authoritative** servers "what's the address?". Your device doesn't do this walk itself; it asks a **recursive resolver** (your ISP's, or a public one like Cloudflare's `1.1.1.1` or Google's `8.8.8.8`), which does the walking and hands back the answer.



The resolver does the hierarchy walk once, then caches the answer for its TTL, so the next lookup of `example.com` is instant. Most DNS traffic never reaches the authoritative servers.

One name, resolved top-down, then cached. The device asks a resolver; the resolver walks root to TLD to authoritative and remembers the answer. Caching at every level is why DNS is usually instant and why changes take time to appear.

EXAMPLE dns.py · the walk, and the cache

```
# DNS resolves a name by walking a hierarchy: root -> top-level domain -> authoritative server.
# A recursive resolver does the walking and caches answers for their TTL, so most lookups are
  free.
ROOT = {"com": "tld-com-server", "org": "tld-org-server"}
TLD = {"tld-com-server": {"example.com": "ns.example.com", "shop.com": "ns.shop.com"}}
AUTH = {"ns.example.com": {"example.com": ("93.184.216.34", 3600)}} # (IP, TTL seconds)

class Resolver:
    def __init__(self):
        self.cache = {} # name -> (ip, expires_at)
        self.queries = 0 # network queries actually made

    def resolve(self, name, now):
        cached = self.cache.get(name)
        if cached and now < cached[1]:
            return cached[0], "CACHE (0 queries)"
        # Walk the hierarchy from the root, counting each network hop.
        tld = "." + name.split(".")[1]
        tld_server = ROOT[tld]; self.queries += 1 # ask the root
        auth_server = TLD[tld_server][name]; self.queries += 1 # ask the TLD
        ip, ttl = AUTH[auth_server][name]; self.queries += 1 # ask the authoritative
            server
        self.cache[name] = (ip, now + ttl)
        return ip, "RESOLVED (root -> TLD -> authoritative, 3 queries)"

if __name__ == "__main__":
    r = Resolver()
    for now in (0, 100, 5000): # TTL is 3600s
        ip, how = r.resolve("example.com", now)
        print(f"t={now:5}s example.com -> {ip} [{how}]")
    print(f"\ntotal network queries for 3 lookups: {r.queries}")
    print("first lookup walks the hierarchy; within the TTL it's cached; after 3600s it re-
        resolves.")
    print("TTL is the trade-off: long = fast & cacheable but slow to change; short = agile but
        chatty.")
```

The first lookup walks the hierarchy (three queries); the second, within the TTL, is served from the resolver's cache with no network at all; after the TTL expires, it re-resolves. Caching happens at every level, the resolver, your OS, your browser, which is why DNS feels instant and why most queries never reach the authoritative servers.

Records, and the TTL trade-off

A name can carry many kinds of record: **A** (IPv4 address), **AAAA** (IPv6), **CNAME** (an alias to another name, how you point a domain at a CDN), **MX** (mail servers), **TXT** (arbitrary text, used for domain verification and email anti-spoofing), and **NS** (the authoritative nameservers). A real query shows a domain often returns *several* A records, which is the simplest load balancing there is: clients spread across the addresses. Every record has a **TTL**, and it's a genuine trade-off: a long TTL (a day) means fast, cacheable, resilient lookups but slow changes, if you need to move a service, the old address lingers in caches

worldwide until the TTL expires; a short TTL (a minute) makes changes and failover quick but multiplies query load and dependence on your DNS being up. Teams routinely *lower* the TTL days before a planned migration so the cut-over is fast, then raise it again.

Why it's always DNS

DNS sits under everything, so when it breaks, healthy services become unreachable and the cause is rarely obvious, hence the operations proverb. The cautionary record is long: in 2016 a DDoS attack on the DNS provider **Dyn** took Twitter, Reddit, Netflix and others offline without touching any of them, because their names couldn't be resolved. In 2021 **Facebook** withdrew the routes to its own DNS servers and vanished for six hours (chapter 14). In 2025 a race condition in **AWS's** automated DNS management briefly emptied the address record for a core DynamoDB endpoint in its busiest region, and the gap cascaded into a 15-hour outage across much of the internet. The pattern repeats: DNS is a critical dependency with no blast-radius limit, its failures look like everyone else's outage, and its automation and caching create subtle, delayed, hard-to-diagnose faults. When something is unreachable, check DNS first, you'll be right more often than you'd like.

Privacy, and the modern layer

DNS was historically sent in **plaintext**, so your ISP, or anyone on the path, could see every domain you looked up, a map of your browsing even over HTTPS. The fix is **encrypted DNS**: **DoH** (DNS over HTTPS, on port 443, blending in with web traffic) and **DoT** (DNS over TLS, on its own port), now a meaningful and growing share of queries and the default in some browsers. There's also **DNSSEC**, which cryptographically signs records so a resolver can detect a forged answer (addressing integrity, not privacy), though its adoption remains small. For a builder, the practical points are: prefer a resolver that supports encrypted DNS, remember that the SNI (chapter 6) still leaked the hostname until ECH, and know that DNS is now one more encrypted, authenticated layer rather than the open postcard it used to be.

COMMON TRAPS

Forgetting the TTL during a migration, so the old address lingers in caches for hours after you switch.

Blaming the application when a service is unreachable, before checking whether its name even resolves.

Treating DNS as instant and infallible, when it's a cached, hierarchical dependency that fails subtly.

A single DNS provider as a single point of failure for your whole presence (the Dyn lesson).

Hardcoding IPs to "avoid DNS", losing the failover and load balancing that multiple records give you

EXERCISES

EXERCISE 15.1 Plan the TTL

You're migrating your website to a new server next Tuesday. Your A record currently has a 24-hour TTL. What do you do, and when, so the switch is fast and clean?

SOLUTION

At least 24 hours (one old TTL) *before* the migration, **lower the TTL** to something short, say 300 seconds. Wait for the old 24-hour TTL to expire everywhere so that resolvers worldwide pick up the short TTL. Now, on Tuesday, change the A record to the new server: because the TTL is 300s, caches refresh within five minutes and traffic cuts over quickly, with no long tail of users stuck on the old server. Keep both servers running during that window to catch stragglers. Once it's stable, raise the TTL back up (to a day) for resilience and lower query load. The key insight: you must lower the TTL far enough ahead that the *old* TTL has drained, or the lowering itself won't have propagated when you need it.

EXERCISE 15.2 Diagnose the "it works for me"

You update your domain to point at a new host. Colleagues in another office still land on the old server hours later, while you see the new one. Nothing is "broken". Why, and how do you confirm it?

SOLUTION

DNS caching. Your resolver (or OS, or browser) has already fetched the new record, while your colleagues' resolver still holds the old one and will until its cached copy's TTL expires. Different resolvers cache independently, so propagation is uneven, not instant. Confirm it with `dig` against different resolvers: `dig @1.1.1.1 yourdomain.com` versus `dig @8.8.8.8 yourdomain.com` will show different answers and different remaining TTLs during propagation. The fix isn't to "force" propagation (you can't reach into everyone's cache); it's patience until the TTL expires, which is exactly why you lower the TTL *before* a change (exercise 15.1). "It works for me" over DNS almost always means a cache hasn't expired.

EXERCISE 15.3 Harden the dependency

Your entire product became unreachable when your DNS provider had an outage, though your servers were fine. What would make you resilient to this?

SOLUTION

- **Use two independent DNS providers** for your authoritative nameservers (DNS supports listing multiple NS records across providers), so one provider's outage doesn't remove you from the internet, the direct lesson of the Dyn attack.
- **Choose sensible TTLs:** not so short that an outage of your DNS immediately breaks everything (longer TTLs let cached answers ride out a brief provider outage), balanced against agility.
- **Monitor resolution from outside** your network and alert on it, so you learn your name stopped resolving before your users tell you.
- **Treat DNS changes like deploys:** review, stage, and have a rollback, because a bad DNS change is as total an outage as a bad routing change (chapter 14).

DNS is infrastructure your whole business depends on; give it the redundancy and care you'd give any single point of failure.

CHECKLIST

- ☐ Can you describe the resolution walk: stub, recursive resolver, root, TLD, authoritative?
- ☐ Do you know the common record types and that multiple A records load-balance?
- ☐ Do you understand the TTL trade-off, and lower it before planned changes?
- ☐ When something's unreachable, do you check DNS early?
- ☐ Is your DNS redundant and monitored, not a hidden single point of failure?

GO DEEPER

COMIC [How DNS Works](#) a friendly illustrated walkthrough

REFERENCE [DNS MDN](#)

PART V

Delivery at scale

Three chapters on making the network fast and highly available for real users. The physics of latency and why distance is the enemy, the content delivery networks and anycast that move servers closer, and the load balancers that spread traffic across them.

Latency, bandwidth and the physics

3 exercises

Latency and bandwidth are different things, and confusing them is the root of most bad performance decisions. Bandwidth you can buy; latency is set by the speed of light and the number of round trips, and no amount of code optimisation or extra bandwidth will beat it. The whole game at this layer is making fewer round trips over shorter distances.

Two numbers, not one

Bandwidth is how much data per second a link can carry; **latency** is how long one trip takes. They're independent: a satellite link can have huge bandwidth and terrible latency, and a short cable the reverse. People say "faster internet" meaning bandwidth, but for most of what you build, loading a page, an API call, a database query, the experience is dominated by *latency*, because the work is a conversation of small messages, each waiting a round trip for its answer. Stuart Cheshire's old essay title says it: "It's the latency, stupid."

Latency has a floor, and it's physics

Light in fibre travels about 200,000 km per second, two-thirds of its speed in vacuum, which is roughly 200 km per millisecond. That sets a hard floor no engineering can cross:

EXAMPLE latency.py · the physics, and the budget

```
# Latency and bandwidth are different resources. Latency is set by distance and the speed of
# light; you cannot optimize code past it. The only way to cut it is fewer, shorter round
# trips.
FIBER_KM_PER_MS = 200          # light in fibre travels ~200 km per millisecond (about 2/3 of
                                # c)

def rtt_ms(distance_km, route_factor=1.5):
    """Round-trip time floor for a distance. route_factor: real fibre isn't a straight line."""
    one_way = distance_km / FIBER_KM_PER_MS * route_factor
    return round(2 * one_way)

def cold_https_ms(rtt):
    """A first visit: DNS + TCP + TLS + the HTTP request, each about one round trip."""
    return 4 * rtt

def transfer_ms(size_kb, rtt, bandwidth_mbps):
    """Total time = latency (round trips) + the bytes / bandwidth."""
    bits = size_kb * 1024 * 8
    download_ms = bits / (bandwidth_mbps * 1_000_000) * 1000
    return cold_https_ms(rtt) + download_ms

if __name__ == "__main__":
```

```

print("round-trip floor by distance (fibre):")
for place, km in [("same city", 50), ("New York-London", 5500), ("New York-Sydney",
16000)]:
    print(f" {place:18} {km:6} km -> ~{rtt_ms(km):3} ms RTT")

print("\na cold HTTPS page, far origin vs nearby CDN:")
far, near = rtt_ms(16000), rtt_ms(50)
print(f" distant origin (RTT {far}ms): {cold_https_ms(far)} ms before the HTML even
starts")
print(f" nearby CDN (RTT {near}ms): {cold_https_ms(near)} ms
({cold_https_ms(far)//cold_https_ms(near)}x faster, same code)")

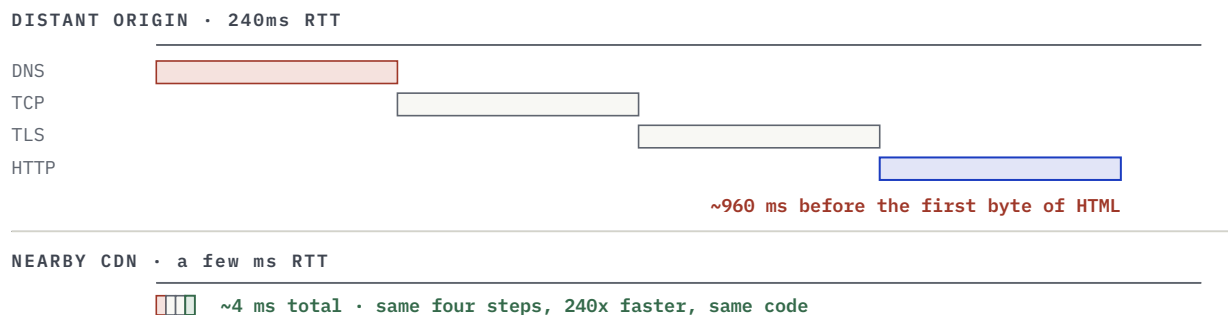
print("\nlatency vs bandwidth: when does each dominate? (RTT 150ms, 100 Mbps)")
for size, label in [(10, "small (10 KB, typical asset)"), (10_000, "large (10 MB,
video)")]:
    total = transfer_ms(size, 150, 100)
    lat = cold_https_ms(150)
    print(f" {label:30} {total:7.0f} ms total, {lat/total*100:3.0f}% is latency")
print("\nfor small objects (most of the web), latency dominates -- more bandwidth won't
help.")

```

New York to Sydney is about 16,000 km, so a round trip can't beat ~160ms in a straight line and is ~240ms over real fibre routes, *per round trip*, forever, no matter how fast the servers or your code. This is the single most important fact at this layer: you cannot optimise your way under the speed of light. You can only make fewer round trips, or make them shorter.

The latency budget: count the round trips

A "slow page" is usually a stack of round trips, each paying that distance tax. A cold HTTPS request is about four round trips before the HTML even begins, DNS, TCP handshake, TLS handshake, then the request itself (chapters 6 and 9), and the example shows what that costs at distance:



The identical request, far and near. Each step is a round trip whose length is distance, so four of them across the world is nearly a second, and four of them to a nearby CDN is a few milliseconds. Shortening the distance (chapter 17) and removing round trips (keep-alive, HTTP/2 and 3, TLS resumption) are the only real levers.

This is why every latency win in this book, connection reuse and keep-alive (chapter 3), HTTP/2 and HTTP/3 multiplexing (chapter 5), TLS resumption and QUIC's 0-RTT (chapters 6, 5), and CDNs (chapter 17), is about the same two moves: **remove round trips, or shorten them**. A "chatty" API that makes ten sequential calls to render a page pays ten times the round trip, which across a continent is the difference between snappy and sluggish, and no faster server fixes it; batching the ten into one does.

When bandwidth matters, and when it doesn't

Bandwidth takes over only for *bulk* transfer. The example's last block makes it concrete: fetching a 10 KB asset at 150ms RTT is essentially all latency (the bytes take under a millisecond to transfer, the round trips take hundreds), so doubling the user's bandwidth changes nothing; fetching a 10 MB video is bandwidth-bound, and there more bandwidth helps directly. Most of the web is small objects, which is why most web performance work is latency work, and why "we upgraded to a gigabit connection and the site is still slow" is such a common, puzzled complaint, the site was never bandwidth-bound. Know which regime you're in before you optimise. And remember the bandwidth-delay product from chapter 10: on a fast, distant link, even bulk transfer is throttled by your window and the RTT, so latency sneaks back in.

Jitter and the tail

Two refinements matter in production. **Jitter** is the variation in latency, round trips aren't all equal, because packets queue behind others (chapter 10's bufferbloat), and jitter is what makes a video call choppy even when average latency is fine. And the number that matters to users isn't the average but the **tail**: the 95th or 99th percentile, the slowest requests, because a user who makes twenty requests to render one page experiences the slowest of the twenty, not the median. Measure and optimise the tail (p95, p99), not the mean, a lesson the reliability and performance stages return to.

COMMON TRAPS

Buying bandwidth to fix a latency problem. For small objects it does nothing.

Chatty protocols: many sequential round trips where one batched call would do.

Ignoring distance: serving a global audience from one region, paying the round trip to the far side of the world.

Optimising the average latency while users feel the tail (p99).

Forgetting the handshake round trips on every new, un-reused connection.

EXERCISE 16.1 Budget the slow page

A page served from a single data centre in Virginia makes, to render, a cold HTTPS connection then 8 sequential API calls, for a user in Singapore (RTT ~230ms). Estimate the time just in round trips. What two changes help most?

SOLUTION

The cold connection is ~4 round trips (DNS, TCP, TLS, first request) ~ 920ms, then 8 sequential API calls add 8 more round trips ~ 1,840ms (they reuse the connection, so no new handshakes, but each still waits a round trip for its answer). That's roughly **2.7 seconds** of pure network latency, before any server or rendering time. The two biggest changes: **batch the 8 calls into 1** (or a few), removing 7 round trips, 1,600ms, at a stroke; and **serve from closer**, a CDN or a Singapore region cutting the RTT from 230ms to maybe 10ms, which shrinks every remaining round trip twenty-fold. Faster server code would barely move the number, because almost none of the 2.7s is server time.

EXERCISE 16.2 Latency or bandwidth?

For each complaint, is the bottleneck latency or bandwidth, and what helps?

- a. A dashboard that makes 30 small API calls feels sluggish on mobile.
- b. Downloading a 4 GB game takes ages.
- c. A video call is choppy though the speed test shows 200 Mbps.
- d. A REST API's single response is fast locally but slow from another continent.

SOLUTION

- a. **Latency** (and round-trip count): 30 small calls each pay a round trip, worse on mobile's higher RTT. Batch them, and use HTTP/2/3 to parallelise.
- b. **Bandwidth**: 4 GB is bulk; here a faster link genuinely helps, as does a nearby mirror/CDN to also shorten the path.
- c. **Latency and jitter**, not bandwidth, 200 Mbps is irrelevant if packets are queueing (bufferbloat, chapter 10). Fix the queueing, not the bandwidth.

- d. **Latency (distance)**: the single round trip is fine locally and expensive across an ocean. A CDN or regional presence shortens it.

EXERCISE 16.3 The impossible requirement

A product manager wants a trading feature where a user in Tokyo sees a confirmation from a server in London within 50ms. Is it possible? What do you tell them?

SOLUTION

No, not as stated. Tokyo to London is roughly 9,500 km; even in a straight line at fibre speed that's ~95ms *round trip*, and real routes push it past 200ms, before the server does any work. A single 50ms round trip between those cities violates physics, so no code, no bandwidth, no vendor can deliver it. What you tell them: the 50ms target is achievable only by **moving the server closer** to the user (a Tokyo region or edge), so the confirming round trip is local; the London server can reconcile asynchronously afterwards. This is exactly why global low-latency systems are built as regional deployments with local confirmation (the distributed-systems stage's home-region pattern), the requirement forces the architecture. Knowing the speed of light turns a vague "make it faster" into a concrete design.

CHECKLIST

- ☐ Do you distinguish latency from bandwidth, and know which bounds your workload?
- ☐ Can you estimate a round trip's floor from distance?
- ☐ Do you count the round trips in an operation, and work to remove them?
- ☐ Do you avoid chatty, sequential calls in favour of batching?
- ☐ Do you measure the tail latency (p95/p99), not just the average?

GO DEEPER

ESSAY It's the Latency, Stupid Stuart Cheshire

BOOK High Performance Browser Networking: latency and bandwidth Ilya Grigorik

REFERENCE Measure performance with the RAIL model web.dev

CDNs, edge and anycast

3 exercises

If latency is set by distance (chapter 16), the way to be fast is to put a copy of your content near every user. That's a content delivery network: hundreds of locations worldwide, serving cached content from close by, reached through anycast, which lets one IP address live in many places and the internet route each user to the nearest. The catch is that a CDN is a shared dependency whose mistakes go global in seconds.

Why a CDN exists

One origin server, however powerful, is far from most of your users, and chapter 16 showed distance is a latency tax no code can dodge. A **content delivery network** solves this by distance: it runs servers, **points of presence** (PoPs), in hundreds of cities, caches your content on them, and serves each user from the nearest one. The round trip that was 240ms to a distant origin becomes a few milliseconds to a local PoP. A CDN does four jobs at once: it **shortens the distance** for latency; it **caches** static content at the edge (chapter 4's caching, now global) so the origin is barely touched; it **absorbs traffic spikes and DDoS** because its edge is vastly larger than your origin; and it **terminates TLS** near the user, so even the handshake round trips are short. Increasingly it also runs your **code** at the edge.

Anycast: one address, many places

How does a user reach the nearest PoP without the application choosing one? The elegant answer is **anycast**: the CDN announces the *same* IP address from every PoP via BGP (chapter 14), and the internet's own routing delivers each user's packets to whichever PoP is nearest in network terms. No DNS trickery, no client logic, routing does it.

EXAMPLE anycast.py · nearest PoP, and automatic failover

```
# A CDN announces ONE IP address from many points of presence (PoPs) at once, via BGP.
# The internet's own routing then delivers each user to the nearest PoP. No DNS trickery
# needed.
POPS = {                                # network "distance" (ms) from each region to each PoP
    "london":    {"eu-user": 8,  "us-user": 75, "asia-user": 160},
    "virginia": {"eu-user": 80, "us-user": 10, "asia-user": 180},
    "singapore": {"eu-user": 170, "us-user": 185, "asia-user": 12},
}

def route(user, live_pops):
    """Anycast sends the user to the PoP that's nearest by network distance (BGP picks it)."""
    return min(live_pops, key=lambda pop: POPS[pop][user])

if __name__ == "__main__":
    users = ["eu-user", "us-user", "asia-user"]
```

```

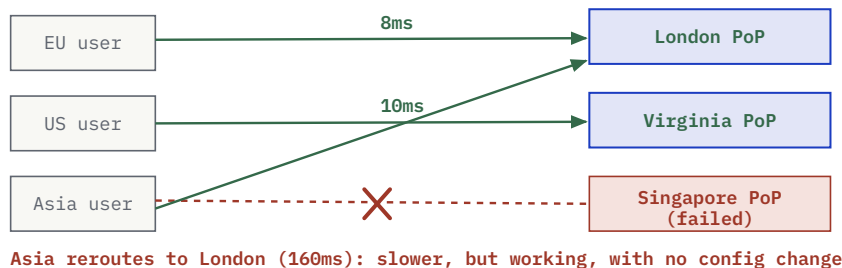
live = list(POPS)
print("normal operation (all PoPs up):")
for u in users:
    pop = route(u, live)
    print(f" {u:10} -> {pop:10} ({POPS[pop][u]} ms)")

print("\nthe Singapore PoP fails; BGP withdraws its route, traffic reroutes
    automatically:")
live_after = [p for p in live if p != "singapore"]
for u in users:
    pop = route(u, live_after)
    print(f" {u:10} -> {pop:10} ({POPS[pop][u]} ms)")
print(" the Asia user now takes a slower but working path -- no config change, just
    routing.")

# Edge caching offloads the origin: only misses travel the long way back.
hit_rate = 0.95
print(f"\nwith a {hit_rate:.0%} edge cache hit rate, the origin sees only "
    f"{(1-hit_rate)*100:.0f}% of requests -- a 20x reduction in origin load.")

```

one IP address (203.0.113.9) announced from every PoP; BGP routes each user to the nearest



Anycast turns failover into routing. All users share one IP; each reaches the nearest PoP. When a PoP dies, BGP stops announcing its route and traffic flows to the next-nearest automatically, the resilience is in the network, not in your application.

The example shows both halves: in normal operation each user reaches their nearest PoP (single-digit milliseconds); when the Singapore PoP fails, its BGP route is withdrawn and the Asia user is rerouted to London, slower, but *working*, with no configuration change and no human action. That automatic rerouting is anycast's quiet superpower, and the edge caching above it means the origin, even in normal times, sees only the small fraction of requests that miss the cache (a 95% hit rate is a twenty-fold reduction in origin load).

The cache hierarchy and the edge

CDNs layer caches: a request hits the nearest **edge** PoP; on a miss it may check a **regional** cache, and only then the **origin** (often through a single "origin shield" PoP, so a cold cache doesn't stampede the origin, the thundering-herd problem the reliability stage covers). Static content (images, scripts, videos) caches beautifully; dynamic, per-user content doesn't, which is why the `private / no-store` directives of chapter 4 matter so much at the edge, a mis-marked personalised response cached at a PoP is served to strangers. The

frontier is **edge compute**: running your code at the PoPs (Cloudflare Workers and the like), so even dynamic logic executes near the user rather than a continent away, blurring the line between "CDN" and "where your app runs".

The shared-dependency risk

A CDN concentrates enormous value and enormous risk. Because the same configuration and code run at every PoP, a bad change propagates *everywhere in seconds*, exactly as fast as the content it's meant to speed up. In 2021 a single customer's valid configuration change triggered a latent bug at **Fastly** and 85% of its network returned errors within a minute, taking down a swathe of the internet, before being fixed in under an hour. In 2025 an oversized internal configuration file crashed **Cloudflare**'s proxies across its global edge, degrading much of the web for hours. The lessons are the ones that recur in this volume: a CDN is a single point of failure for everything behind it, so its own changes need staging and rollback like any deploy; anycast spreads a fix globally but spreads a bug just as globally; and depending on one provider for your entire presence is a risk worth weighing against the very real benefits. The benefits usually win, but go in with eyes open.

COMMON TRAPS

Caching personalised responses at the edge, serving one user's data to another (chapter 4's `private`).

Expecting the CDN to speed up dynamic, per-user content it can't cache, use edge compute or accept the origin round trip.

Forgetting the origin still needs protection: cache misses, purges and cold caches can stampede it.

Treating the CDN as infallible, when its global reach means its outages are your global outage.

No origin failover plan for when the CDN itself is the problem.

EXERCISES

EXERCISE 17.1 What belongs on the CDN?

For each, should it be served/cached by the CDN edge, run at the edge, or go to the origin?

- Product images and the site's JavaScript bundle.
- A logged-in user's personalised dashboard HTML.
- A simple A/B-test redirect based on a cookie.

d. Placing an order (a payment).

SOLUTION

- a. **Cache at the edge**, with long TTLs and content-hashed URLs (chapter 4). Static, shared, perfect for the CDN.
- b. **Origin** (or edge-rendered carefully), and marked `private` / `no-store` so no shared cache holds it. Per-user HTML must never be edge-cached for everyone.
- c. **Edge compute**. A tiny, latency-sensitive decision based on a cookie is ideal to run at the PoP, avoiding an origin round trip.
- d. **Origin**. A payment is dynamic, stateful and must not be cached; the CDN just passes it through (terminating TLS and absorbing abuse on the way).

EXERCISE 17.2 Reason about anycast failover

With anycast, when a PoP fails, users on it reroute to the next-nearest automatically. What's the upside, and what's a subtle downside for a *stateful* connection like a long download or a WebSocket?

SOLUTION

The upside is resilience with no moving parts: failover is just BGP reconverging, no health-check-and-reconfigure loop, no DNS TTL to wait out, traffic simply flows to the next PoP. The subtle downside is that anycast routes *packets*, and routing can change mid-connection: if the network re-converges (a link flaps, a PoP's announcement changes) while you have a long-lived TCP connection, your packets may suddenly arrive at a *different* PoP that has no state for that connection, and it resets. This is why anycast is perfect for short, stateless request/response (DNS, HTTP fetches) but stateful long connections need care, the PoPs must share state, or the connection must tolerate being re-established (chapter 7's reconnect logic), or you use a protocol like QUIC whose connection ID survives a path change (chapter 5). The same property that makes failover automatic can break a connection that assumed a fixed endpoint.

EXERCISE 17.3 Weigh the dependency

Your CEO reads about the Cloudflare and Fastly outages and asks whether using a CDN is too risky. What's your balanced answer?

SOLUTION

The honest answer: a CDN is both a huge net win and a concentrated risk, and the right response is to manage the risk, not avoid the benefit. The win is real and daily, lower latency for every user, massive origin offload, DDoS protection, free TLS termination, which a single origin can't match, and running your own global edge is wildly impractical for most teams. The risk is that the CDN becomes a single point of failure whose rare outages are total. Mitigations: pick a reputable provider with a good track record; keep the ability to **fail over to the origin or a second CDN** (multi-CDN, often via DNS) for critical paths; don't build logic that *only* works through the CDN; and monitor from outside. For most organisations the CDN's outages are rarer and shorter than the outages you'd suffer serving a global audience from one origin, so it's a risk you take deliberately, with a fallback, rather than one you refuse.

CHECKLIST

- ☐ Do you understand a CDN's four jobs: distance, caching, absorption, TLS termination?
- ☐ Do you know anycast routes each user to the nearest PoP and fails over via BGP?
- ☐ Do you mark personalised responses so they're never edge-cached?
- ☐ Is the origin protected from cache misses and stampedes?
- ☐ Do you have a plan for when the CDN itself fails?

GO DEEPER

REFERENCE [What is a CDN? and anycast Cloudflare Learning](#)

REPORT [Cloudflare's November 2025 outage post-mortem Cloudflare](#)

BOOK [High Performance Browser Networking Ilya Grigorik](#)

Load balancing at layers 4 and 7

3 exercises

One server can't take all the traffic, and no single server should be a single point of failure, so a load balancer sits in front of a pool and spreads requests across it. Where it sits in the stack decides what it can do: a layer-4 balancer routes whole connections by their 4-tuple and is blind to HTTP; a layer-7 balancer reads each request and routes on its path, headers and health. The one thing both must do is stop sending traffic to a server that's down.

Two places to balance

Everything in this volume has built toward a request arriving at a server. But "a server" is a lie: behind one address is a *pool* of identical servers, and a **load balancer** (LB) decides which one handles each request. It does two jobs that matter equally: it **spreads load** so no server is overwhelmed, and it **provides availability** by routing around servers that fail, so one dead instance is invisible to users. Where the LB operates in the layered model (chapter 1) decides what it can see and do.

A **layer-4 load balancer** works at the transport layer. It sees TCP connections, the 4-tuple from chapter 9, and nothing above: it can't read the URL or headers, because those live inside the byte stream (and under TLS, they're encrypted). So it picks a backend *per connection*, typically by hashing the 4-tuple so a given client sticks to one backend, and shovels bytes through. It's simple, extremely fast, and protocol-agnostic, it balances anything over TCP, not just HTTP.

A **layer-7 load balancer** terminates the connection, reads the HTTP request, and routes *per request* on what it can now see: path, Host, headers, cookies. It can send `/api/*` to one pool and `/static/*` to another, retry a failed request on a different backend, add tracing headers, enforce rate limits, and do TLS termination. It's doing far more work, so it's slower per request than L4, but it's where almost all modern application routing happens.

EXAMPLE `lb.py` · L4 by connection, L7 by request, both with health checks

```
import zlib

# -----
# A layer-4 load balancer works at the transport layer. It sees a TCP
# connection -- the 4-tuple (src ip, src port, dst ip, dst port) -- and nothing
# above it. It can't read the URL or headers (that's inside the encrypted,
# or at least higher-layer, byte stream). So it picks a backend per CONNECTION
# and sends the whole connection there, usually by hashing the 4-tuple so the
# same client sticks to the same backend.
def l4_route(four_tuple, backends):
    h = zlib.crc32(four_tuple.encode())
    return backends[h % len(backends)]
```

```

backends = ["app-a", "app-b", "app-c"]
conns = ["10.0.0.5:51000->lb:443", "10.0.0.5:51001->lb:443", "10.0.0.9:62000->lb:443"]
print("layer 4 (routes whole connections by 4-tuple hash):")
for c in conns:
    print(f" {c:32} -> {l4_route(c, backends)}")

# -----
# A layer-7 load balancer terminates the connection, reads the HTTP request,
# and routes PER REQUEST on what it can see: the path, the Host, headers,
# cookies. It can send /api to one pool and /img to another, retry a failed
# request on another backend, and add/inspect headers.
POOLS = {"api": ["api-1", "api-2"], "static": ["cdn-1", "cdn-2"]}

def l7_route(method, path, healthy):
    pool = "api" if path.startswith("/api/") else "static"
    live = [b for b in POOLS[pool] if healthy.get(b, True)]
    if not live:
        return pool, None # no healthy backend -> 503
    # least-connections would go here; round-robin by a tiny counter instead
    idx = l7_route.rr.setdefault(pool, 0)
    l7_route.rr[pool] = (idx + 1) % len(live)
    return pool, live[idx]
l7_route.rr = {}

print("\nlayer 7 (routes each request by path, across healthy backends):")
healthy = {"api-1": True, "api-2": True, "cdn-1": True, "cdn-2": True}
reqs = [("GET", "/api/orders"), ("GET", "/api/users"), ("GET", "/img/logo.png"), ("GET",
"/img/hero.jpg")]
for m, p in reqs:
    pool, b = l7_route(m, p, healthy)
    print(f" {m} {p:16} -> {pool} pool -> {b}")

# -----
# Health checks: the LB probes each backend and stops sending to one that
# fails. This is the whole point -- a dead backend is routed around in seconds,
# with no request landing on it, no user noticing.
print("\napi-2 fails its health check; the LB routes around it:")
healthy["api-2"] = False
l7_route.rr = {}
for m, p in [("GET", "/api/a"), ("GET", "/api/b"), ("GET", "/api/c")]:
    pool, b = l7_route(m, p, healthy)
    print(f" {m} {p:10} -> {b}")
assert all(l7_route(m, p, healthy)[1] == "api-1" for m, p in [("GET", "/api/x")])
print(" every /api request now goes to api-1; api-2 gets zero traffic until it recovers.")

```

Notice the difference the layer makes. The L4 balancer sends two connections *from the same client* (ports 51000 and 51001) to different backends, it balances connections, not requests, and knows nothing about what's inside. The L7 balancer splits traffic by path into separate pools and, crucially, routes every request *around* the backend that failed its health check. That health check is the heart of availability: the LB probes each backend continuously and, the moment one stops answering, stops sending it traffic, so a crash or a bad deploy takes a server out of rotation in seconds with no user seeing an error.

Choosing a backend, and keeping sessions

Within a healthy pool, the LB needs a policy. **Round-robin** cycles through backends; it's fine when requests cost about the same. **Least-connections** sends each new request to the backend with the fewest in flight, which handles uneven request costs far better and is a

good default. **Power of two choices**, which the distributed-systems volume covered, picks two backends at random and takes the less loaded, getting most of least-connections' benefit without global state. For caches and sharded backends, **consistent hashing** maps each key to a backend so that adding or removing one backend remaps only a small fraction of keys.

A recurring wrinkle is **session affinity** ("sticky sessions"): routing a given user to the same backend so in-memory session state is there. It works, but it undermines balancing (a popular user overloads one backend) and breaks when that backend dies (the session's gone). The better answer is almost always to make backends **stateless**, holding session state in a shared store (chapter on caching in the data volume, or a database), so *any* backend can serve *any* request and the LB is free to balance purely on load. Statelessness is what makes horizontal scaling and painless failover possible.

COMMON TRAPS

No health checks, so the LB keeps sending requests to a dead or broken backend.

Health check too shallow (a TCP connect, or a 200 from a trivial endpoint) so it passes while the app is actually broken, check something real.

Relying on sticky sessions instead of stateless backends, so failover loses user state and load skews.

Expecting an L4 balancer to route by URL or retry individual requests, it can't see them.

The load balancer itself as a single point of failure, with no redundant LB (and anycast or DNS in front).

EXERCISES

EXERCISE 18.1 L4 or L7?

For each need, which layer of load balancer do you reach for?

- Route `/api/*` to one pool of servers and everything else to another.
- Balance a pool of database replicas speaking a custom TCP protocol.
- Retry a failed request on a different backend transparently.
- Maximum throughput for millions of long-lived connections, with no per-request logic.

SOLUTION

- a. L7. Routing by path requires reading the HTTP request, only a layer-7 balancer can.
- b. L4. A layer-4 balancer is protocol-agnostic, it balances any TCP stream, not just HTTP, which is exactly what a custom protocol needs.
- c. L7. Retrying an individual request means the LB must understand request boundaries and idempotency, which lives at layer 7. (An L4 balancer only sees a connection; if it breaks, the whole connection breaks.)
- d. L4. When there's no per-request decision to make, the transport-layer balancer's simplicity and speed win, it just forwards bytes.

EXERCISE 18.2 The sticky-session trap

A team uses sticky sessions because each backend keeps the user's shopping cart in memory. During a deploy, one backend restarts and users lose their carts. What's the fix, and why is it better than "just make stickiness more reliable"?

SOLUTION

The fix is to make the backends **stateless**: move the cart out of any single server's memory into a shared store (a cache like Redis, or the database) keyed by session or user. Then any backend can serve any request, so a restart, a crash, or a scale-down loses nothing, and the load balancer can balance purely on load instead of being pinned by affinity. This is better than hardening stickiness because stickiness fights the two things you actually want: even load (a heavy user overloads their one backend) and painless failover (a dead backend's sessions are simply gone, no matter how reliable the routing was). You can't make "this user's state lives on exactly one server" survive that server dying, the only real fix is to stop depending on one server. Statelessness is the property that makes horizontal scaling and rolling deploys safe.

EXERCISE 18.3 A health check that lies

An LB's health check does a TCP connect to each backend's port. A backend's process is up and accepting connections, but its database pool is exhausted so every real request returns 500. What happens, and what should the health check do instead?

SOLUTION

The TCP connect succeeds, so the LB believes the backend is healthy and keeps sending it traffic, every request of which fails with a 500. A shallow check (can I open a socket? does `/ping` return 200 from a handler that does nothing?) measures the wrong thing: it confirms the process is alive, not that it can actually do its job. The health check should exercise a real dependency path, a `/health` (or readiness) endpoint that checks the things a real request needs: can it get a database connection, reach its critical downstreams, etc., and returns unhealthy when it can't. Then the LB pulls this backend out of rotation and routes around it, exactly as the example did for the failed backend. The balance to strike: deep enough to catch real breakage, cheap enough to run every few seconds, and careful not to cascade (a shared-dependency outage shouldn't mark *every* backend unhealthy at once and leave zero in rotation).

CHECKLIST

- ☐ Do you know L4 routes connections by 4-tuple and L7 routes requests by their content?
- ☐ Does every backend have a health check that exercises something real?
- ☐ Are your backends stateless, so any one can serve any request?
- ☐ Have you chosen a balancing policy (least-connections is a good default)?
- ☐ Is the load balancer itself redundant, not a new single point of failure?

GO DEEPER

REFERENCE What is load balancing? NGINX

PAPER The power of two random choices Mitzenmacher

BOOK Load balancing in the datacenter Google SRE Book

PART VI

Operating the network

Two chapters on living with the network you've built. How to diagnose a problem by localising it to one layer before you touch anything, and a final walk end to end, from a typed URL to painted pixels, that ties every chapter of this volume into a single request.

Diagnosing the network

3 exercises

"It's slow" and "it doesn't work" are not diagnoses, they're symptoms that could live at any layer from DNS to the application. The skill is to localise the problem to one layer before you try to fix it, and the whole layered model of this volume is the method: walk down the stack, asking at each layer whether it's the problem, using the one tool that answers that question. Most of the time, the answer surprises you.

The method: localise before you fix

The commonest mistake in debugging a network problem is to guess a cause and start changing things. The disciplined approach is the one this whole volume has been building toward: the stack is layered, so *diagnose by layer*. A request's life is name → connection → TLS → request → response (the chapters in order), and a failure or slowness lives at exactly one of those. Find *which* first; the fix follows from the layer.

The single most useful trick is to ask **where the time goes**. `curl` will tell you, phase by phase, and because the phases are the layers, the breakdown points straight at the culprit.

EXAMPLE `diag.py` · the same symptom ("slow") localised to three different layers

```
# curl reports cumulative timing checkpoints with -w. Each is seconds since the
# request began, so the TIME IN EACH PHASE is the gap between checkpoints. This
# is the single most useful network-diagnosis trick: it tells you WHICH LAYER
# is slow, so you debug the right thing instead of guessing.
#
# curl -w "@curl-format.txt" -o /dev/null -s https://example.com
#
# with a format file printing time_namelookup, time_connect, time_appconnect,
# time_starttransfer and time_total.
```

```
PHASES = [
    # (label, which layer, which tool to reach for next)
    ("DNS lookup", "naming (ch15)", "dig / resolvectl"),
    ("TCP connect", "transport (ch9)", "traceroute / ss"),
    ("TLS handshake", "security (ch6)", "openssl s_client"),
    ("server thinking", "the application", "server logs / APM"),
    ("download", "the response/path", "check payload size, bandwidth"),
]
```

```
def breakdown(t_dns, t_connect, t_appconnect, t_ttfb, t_total):
    # cumulative checkpoints -> per-phase durations
    durs = [
        t_dns, # DNS
        t_connect - t_dns, # TCP handshake
        t_appconnect - t_connect, # TLS handshake
        t_ttfb - t_appconnect, # waiting for first byte (server work)
        t_total - t_ttfb, # downloading the body
    ]
    return durs
```

```
def diagnose(name, *checkpoints):
    durs = breakdown(*checkpoints)
    total = checkpoints[-1]
    print(f"{name} (total {total*1000:.0f} ms)")
    worst_i, worst = 0, -1.0
    for i, (dur, (label, layer, tool)) in enumerate(zip(durs, PHASES)):
        pct = dur / total * 100
        bar = "#" * round(pct / 5)
        print(f" {label:16} {dur*1000:6.0f} ms {pct:4.0f}% {bar}")
        if dur > worst:
            worst_i, worst = i, dur
    label, layer, tool = PHASES[worst_i]
    print(f" -> dominated by {label}: look at {layer}; next tool: {tool}\n")
    return label

# Scenario A: a slow page. Where's the time? (cumulative seconds)
a = diagnose("A: 'the site is slow'", 0.004, 0.028, 0.072, 0.840, 0.880)
# Scenario B: same site, different user, 'slow' again -- but a different cause.
b = diagnose("B: 'same site, also slow'", 0.310, 0.335, 0.380, 0.405, 0.430)
# Scenario C: fast first byte, slow overall -- it's the download, not the server.
c = diagnose("C: 'slow to finish'", 0.005, 0.020, 0.050, 0.070, 1.900)

assert a == "server thinking" # app is slow -> look at the server, not the network
assert b == "DNS lookup"     # 310ms resolving -> a DNS problem, not the app
assert c == "download"       # server was fast; it's a fat payload or a slow path
print("same symptom ('slow'), three different layers -- the breakdown tells them apart.")
```

Three users all say "it's slow". The timing breakdown shows they have three unrelated problems: one is the application taking 768ms to produce a byte (look at server logs, not the network); one is spending 310ms in DNS (a resolver problem, chapter 15); one has a fast server but a 1.8-second download (a fat payload or a congested path). Without the breakdown you'd guess; with it, you know which layer to open next. This is not a toy, it's exactly what `curl` gives you in real life:

REAL CAPTURE `curl -w timing against https://example.com`

```
$ curl -sS -o /dev/null \
-w 'dns=%{time_namelookup} connect=%{time_connect} tls=%{time_appconnect} ttfb=%
{time_starttransfer} total=%{time_total}\n' \
https://example.com

dns=0.049813 connect=0.125818 tls=0.214593 ttfb=0.297702 total=0.298189
# per-phase: DNS 50ms, TCP 76ms, TLS 89ms, server 83ms, download <1ms
# http=2 code=200 ip=2606:4700:10::ac42:93f3 (HTTP/2, over IPv6)
```

A tool for each layer

Each layer has a tool that answers "is it you?". Learn the one question each answers, not every flag:

LAYER	QUESTION	TOOL
Naming (ch15)	Does the name resolve, to what, how fast?	<code>dig</code> , <code>nslookup</code> , <code>resolvectl</code>

Reachability (ch13-14)	Can packets get there, and by what path?	<code>ping</code> , <code>traceroute</code> / <code>mtr</code>
Transport (ch9)	What connections exist, in what state?	<code>ss</code> (or <code>netstat</code>), <code>ss -tan</code>
TLS (ch6)	Is the certificate and handshake good?	<code>openssl s_client</code>
HTTP (ch3-5)	What exactly did the server send back?	<code>curl -v</code> , browser DevTools
The wire (all)	What packets actually crossed the link?	<code>tcpdump</code> , Wireshark

Two reach deeper when the high-level tools aren't enough. `curl -v` shows the whole exchange, DNS result, the IP it connected to, the TLS certificate, every request and response header, which resolves most "why is this response weird?" questions on its own. And when you genuinely need the ground truth, `tcpdump` (or Wireshark) captures the actual packets, the only tool that can't lie to you, because it shows what crossed the wire rather than what some layer *says* happened. Reach for it when the layers disagree: the app says it sent the request, the server says it never arrived, `tcpdump` on both ends settles it. The order is the lesson: start high and cheap (`curl -w` , DevTools), go low and expensive (`tcpdump`) only when you must.

Read the DevTools waterfall

In the browser, the Network panel is this same breakdown made visual: each request is a bar split into DNS, connect, TLS, "waiting" (TTFB) and download, exactly the phases above. A long "waiting" segment is a slow server; a long "download" is a fat response; a wall of requests stacked in sequence (rather than overlapping) is a protocol or dependency problem (chapter 5's head-of-line blocking, or render-blocking resources). The waterfall is where the whole volume becomes visible in one screen, and learning to read it is the most transferable network skill a web engineer has.

COMMON TRAPS

Fixing before localising, changing config at a layer that was never the problem.

Trusting a layer's own report instead of observing the wire when layers disagree (that's what `tcpdump` is for).

Pinging a host and declaring the network fine, when ICMP is often treated differently from your actual TCP traffic (and may be blocked entirely).

Ignoring DNS caching, so `dig` shows a fresh answer while your app holds a stale cached one (chapter 15)

Testing from your machine only, when the problem is regional, test from where the users are (another network, a cloud box, a probe).

EXERCISES

EXERCISE 19.1 Pick the tool

For each symptom, what's the first command you run?

- "The site works for me but not for my colleague in another office."
- "Requests to our API hang for 30 seconds, then succeed."
- "Users get a certificate warning."
- "The page loads but one image is broken."

SOLUTION

- `dig` from both locations (does the name resolve to the same IP?) and `tracert` from the colleague's network. A difference by location is usually DNS or routing, not the app.
- `curl -w` to see which phase hangs. A 30-second hang that then succeeds smells like a connection or DNS timeout-then-retry, the breakdown shows whether it's `connect` (reachability/firewall) or `namelookup` (a dead resolver being retried).
- `openssl s_client -connect host:443` (or `curl -v`) to see the certificate, its chain, expiry, and the name it's issued for, chapter 6's whole checklist in one command.
- `curl -v` (or DevTools Network) on the image's URL, to see the exact status and headers. A 404, a 403, a wrong `Content-Type`, or a CORS/mixed-content block will be right there in the response.

EXERCISE 19.2 Read a breakdown

A request shows: `namelookup=0.004 connect=0.009 appconnect=0.014 starttransfer=2.010 total=2.040` (seconds). Which layer is the problem, and which layer is *not*?

SOLUTION

Compute the per-phase gaps: DNS 4ms, TCP connect 5ms, TLS 5ms, then **starttransfer – appconnect** = ~2 seconds waiting for the first byte, then 30ms to download. The network layers are all healthy, DNS, TCP and TLS together took 14ms. The problem is entirely **server think-time** (TTFB): the request reached the server fast, but the application took two seconds to produce the first byte. So you do *not* touch DNS, the connection, or TLS; you look at the application, slow query, lock contention, a blocking downstream call, cold cache. The breakdown has saved you from "optimising the network" when the network was never the issue.

EXERCISE 19.3 When the layers disagree

Your service logs say it sent a response. The client logs say it received nothing and timed out. Both teams are sure they're right. How do you settle it, and what are the likely answers?

SOLUTION

Capture the actual packets at both ends with `tcpdump` (filtered to the 4-tuple, chapter 9) at the same time, because the packets are the one source of truth that neither side's application logging can spin. Then compare. Likely findings: (a) the server sent the response but it never left its host, or was dropped by a firewall/security group in between, you'll see the server's outbound packet and no corresponding arrival at the client; (b) the response arrived but the client's connection had already timed out and closed, so it was RST, you'll see the bytes arrive followed by a reset; (c) an MTU/fragmentation black hole, large packets silently dropped while the handshake (small packets) succeeded, a classic "small requests work, big responses hang". In every case, `tcpdump` turns "we both think we're right" into "here's the packet where it went wrong", which is why it's the diagnosis of last resort and the one that ends arguments.

CHECKLIST

- ☐ Do you localise to a layer before changing anything?
- ☐ Do you reach for `curl -w` first to see where the time goes?
- ☐ Do you know the one tool that answers each layer's question?
- ☐ Can you read a DevTools waterfall and name what each segment means?
- ☐ Do you know to drop to `tcpdump` when the layers disagree?

GO DEEPER

REFERENCE [Measuring performance & the Network panel MDN](#)

GUIDE [Systems performance & the USE method Brendan Gregg](#)

TOOL [Wireshark User's Guide Wireshark](#)

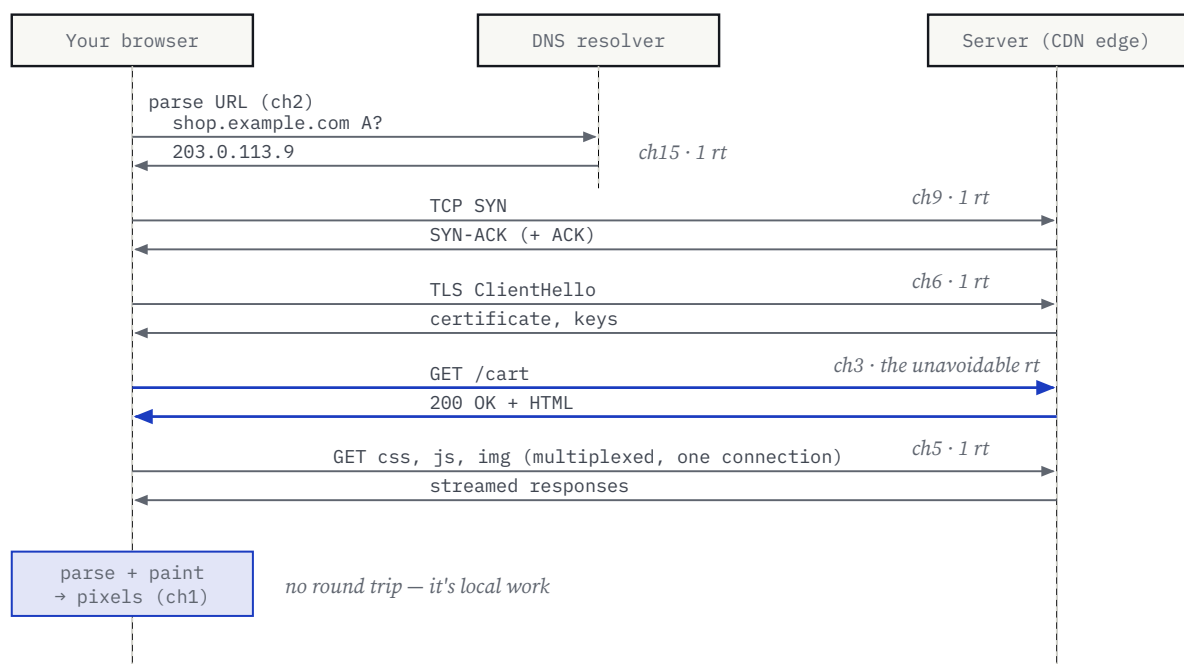
What happens when you type a URL

3 exercises

The classic interview question is really a request to narrate this entire volume as one story. Type a URL, press Enter, and in a few hundred milliseconds a dozen chapters happen in order: the URL is parsed, the name resolved, a connection opened, encryption negotiated, a request sent, a response streamed, and pixels painted. Understanding it end to end is understanding that almost every step is a round trip, and that making the web fast is making fewer of them over shorter distances.

The whole journey, in order

Here is the request that this volume has been assembling, chapter by chapter, now run start to finish. Each step is tagged with the chapter that explains it, so this doubles as the index to everything before it.



One request, the whole volume. DNS resolves the name (ch15); the TCP (ch9) and TLS (ch6) handshakes build the secure channel; the HTTP exchange (ch3, highlighted) is the one round trip no optimisation removes; sub-resources share the connection via multiplexing (ch5); the browser paints (ch1). Five round trips the first time; the warm repeat skips DNS, TCP and TLS, leaving two.

Counting the cost, cold and warm

Narrating the steps is half the answer; the senior half is knowing *what each step costs*. Almost every step is a round trip, and a round trip's price is distance (chapter 16). So the same journey costs wildly different amounts depending on whether you've been here before.

EXAMPLE url.py · the journey priced in round trips, cold vs warm

```
# What actually happens when you type a URL and press Enter? Every chapter of
# this volume, in order, as one request. Here it is as a timeline, each step
# tagged with the round trips it costs -- because round trips x distance is the
# whole latency story (ch16). We price it two ways: a COLD first visit (every
# handshake from scratch) and a WARM repeat (connection reuse, DNS cached).

RTT_MS = 40 # a realistic round trip to a nearby CDN edge (ch16, ch17)

# (step, chapter, round trips when cold, round trips when warm)
STEPS = [
    ("Parse the URL into scheme/host/path",      "ch2",  0, 0),
    ("Resolve the host with DNS",                 "ch15", 1, 0), # cached on repeat
    ("Open a TCP connection (3-way handshake)",   "ch9",  1, 0), # reused on repeat
    ("TLS 1.3 handshake",                        "ch6",  1, 0), # resumed/0-RTT on repeat
    ("Send the HTTP request, get first byte",     "ch3",  1, 1), # the one unavoidable trip
    ("Server does its work (DB, etc.)",           "ch18", 0, 0), # not a network trip
    ("Download HTML; multiplex sub-resources",    "ch5",  1, 1), # H2/H3: one more, overlapped
    ("Browser parses and paints",                 "ch1",  0, 0),
]

def total(which):
    i = 2 if which == "cold" else 3
    trips = sum(s[i] for s in STEPS)
    return trips, trips * RTT_MS

print(f"typing https://shop.example.com/cart (RTT = {RTT_MS} ms)\n")
print(f"{'step':42} {'chapter':7} {'cold':>6} {'warm':>6}")
for name, ch, cold, warm in STEPS:
    print(f"{'name':42} {'ch':7} {'cold':>5}rt {'warm':>5}rt")

ct, cms = total("cold")
wt, wms = total("warm")
print(f"\ncold first visit: {ct} round trips ~={cms} ms of network latency")
print(f"warm repeat:      {wt} round trips ~={wms} ms (DNS cached, connection reused, TLS resumed)")
print(f"the optimisations in this book removed {ct-wt} of {ct} round trips -- a {cms/wms:.1f}x cut in latency.")

# The single most important fact: almost every step is a ROUND TRIP, and a round
# trip's cost is distance (ch16). You make a page fast by making fewer of them
# (keep-alive, H2/H3 multiplexing, caching, ch3-5) over shorter ones (CDN
# edges and anycast, ch16-17). That's the whole volume in one sentence.
assert (ct, cms) == (5, 200)
assert (wt, wms) == (2, 80)
print("\nfewer round trips, over shorter distances. That is the entire game.")
```

The cold first visit pays for everything: DNS, TCP, TLS, then the request and its sub-resources, five round trips, 200ms of pure network latency before anything useful. The warm repeat skips three of them, DNS is cached (chapter 15), the connection is reused (chapter 9, keep-alive), TLS is resumed or 0-RTT (chapter 6), leaving just the two round trips

to fetch content. Every optimisation in this volume is an attack on that count: keep-alive and multiplexing remove connection round trips (chapters 3, 5); caching removes whole requests (chapter 4); CDNs and anycast shrink the distance each remaining round trip travels (chapters 16, 17); HTTP/3 folds the transport and TLS handshakes together (chapter 5). They are all the same move: **fewer round trips, over shorter distances**.

The volume in one breath

If you can tell this story, you understand the web's networking. A URL names a resource (ch2) reached over a layered stack where each layer wraps the one above (ch1). DNS turns the name into an address (ch15); packets find their way there across networks joined by routers, through NAT, over IP (ch13, ch14). TCP builds a reliable ordered byte stream on top of unreliable packets with handshakes, ACKs and congestion control (ch9, ch10); TLS makes it private and authenticated (ch6). On that channel, HTTP carries requests and responses (ch3), made fast by caching (ch4) and by multiplexing over HTTP/2 and HTTP/3 (ch5), and made live by WebSockets and the rest (ch7). Scale comes from CDNs, anycast and load balancers putting copies close and spreading the load (ch16, ch17, ch18); and when it breaks, you localise the fault to one layer and fix it there (ch19). All of it in the time it takes to blink, if you've kept the round trips few and the distances short.

COMMON TRAPS

Narrating steps without costs, listing DNS, TCP, TLS, HTTP but not knowing which are round trips and which are cached away.

Forgetting the warm path, optimising a first visit that most users never experience, instead of the repeat visit they do.

Thinking rendering is network, parse and paint are local CPU work, not a round trip (though the bytes that feed them were).

Stopping at "it returns HTML," when the sub-resource fetches (ch5) are where real page-load time is spent.

Treating the layers as separate, when the whole point is that they compose into one request.

EXERCISES

EXERCISE 20.1 Tell the whole story

Without looking back, narrate what happens from pressing Enter on `https://shop.example.com/cart` to seeing the page, naming each round trip and whether it's skippable on a repeat visit.

SOLUTION

Parse the URL into scheme/host/path (local, no trip). **DNS:** resolve `shop.example.com` to an IP, one round trip, *cached* on repeat (ch15). **TCP:** three-way handshake to the server (via its CDN edge), one round trip, *reused* on repeat via keep-alive (ch9). **TLS 1.3:** handshake to establish encryption, one round trip, *resumed or 0-RTT* on repeat (ch6). **HTTP:** send `GET /cart`, server does its work, first byte arrives, one round trip, *unavoidable*, you always pay for the actual content (ch3). **Sub-resources:** the HTML references CSS/JS/images, fetched multiplexed over the same connection (ch5), one more effective round trip, reducible by caching them (ch4). **Render:** parse and paint, local work (ch1). Cold: five round trips. Warm: two. The ones you skip are exactly the handshakes, which is why connection reuse and TLS resumption matter so much.

EXERCISE 20.2 Make it fast

The page above takes 200ms of network latency on a cold visit. Your users are global and mostly first-time. Name three changes, each mapped to a chapter, and what each one removes.

SOLUTION

Several good answers; three high-impact ones: (1) **Put a CDN in front** (ch16–17): this shrinks the *distance* of every single round trip, so all five cost less, a global first-time user goes from a 150ms RTT to a continent-local 20ms edge, cutting the 200ms dramatically with no code change. (2) **Serve over HTTP/3** (ch5): QUIC folds the transport and TLS handshakes into effectively one round trip (and 0-RTT on repeat), removing a whole round trip from the cold path, and it survives network changes (ch5). (3) **Cache aggressively and version by content hash** (ch4): make sub-resources cacheable forever so repeat visits skip those fetches entirely, and set up the first response to let the browser reuse the connection (keep-alive, ch3). Note how each maps to the single lever, fewer round trips (HTTP/3, caching) over shorter distances (CDN), which is the whole volume's thesis.

EXERCISE 20.3 Where did it break?

A user reports the page "just spins forever". Walk the same journey as a *diagnosis*: at each step, what would you check, and what failure there looks like "spinning"?

SOLUTION

Walk the layers in order (ch19's method). **DNS**: does the name resolve (`dig`)? A dead resolver or wrong record means the browser never gets an IP, pure spin before any connection. **TCP**: can you connect to the IP on 443 (`curl -v`, `tracert`)? A firewall/security-group drop or an unreachable host makes `connect` hang, spin with no response. **TLS**: does the handshake complete (`openssl s_client`)? A cert or protocol mismatch can hang or warn. **HTTP**: does the server return a first byte (`curl -w TTFB`)? A slow or deadlocked app spins after connecting, this is the commonest real cause, and the timing breakdown pins it immediately. **Sub-resources**: does the HTML arrive but a blocking script 404 or hang (DevTools waterfall)? The page can "spin" because one render-blocking resource never completes. The discipline is the same as the happy path: the journey is layered, so localise to the layer that's hanging before you touch anything.

CHECKLIST

- ☐ Can you narrate URL → pixels naming every step and its chapter?
- ☐ Do you know which steps are round trips and which are local?
- ☐ Can you say what the warm path skips, and why?
- ☐ Can you map each optimisation to "fewer round trips or shorter distances"?
- ☐ Can you run the same journey backwards as a diagnosis?

GO DEEPER

CLASSIC What happens when... (the full walkthrough) Alex Gaynor et al.

BOOK High Performance Browser Networking Ilya Grigorik

BOOK Computer Networking: A Top-Down Approach Kurose & Ross

One per chapter, each linked back to it. Read them before you put a service on the network, and against every change that touches a request, a connection or a name.

-
- 01 Think in layers: every message is wrapped by the layer below and unwrapped by its peer, so debug at the layer that owns the problem.
-
- 02 Treat the URL as structured identity, not a string; the origin, scheme plus host plus port, is the web's security boundary.
-
- 03 Know how a response is framed, Content-Length or chunked, because getting the byte count wrong corrupts the stream or hangs the connection.
-
- 04 Cache deliberately with explicit directives and a validator; an unversioned, wrongly-cached response is a bug you can't reach to fix.
-
- 05 Multiplex over one connection, but remember HTTP/2 still rides one TCP stream a single lost packet can stall, and HTTP/3 is the fix.
-
- 06 Let TLS prove identity and encrypt; verify the whole certificate chain to a trusted root, and never click past a certificate error.
-
- 07 Pick the lightest real-time transport that fits, and build reconnection and backpressure in from the first line.
-
- 08 Treat the schema as a contract: field numbers are identity, so add fields, never renumber or reuse them.
-
- 09 Trust TCP for an ordered, reliable byte stream, but remember it's a stream, not messages, you must frame your own.
-
- 10 Respect congestion control: throughput is bounded by the bandwidth-delay product, so a fat pipe far away still needs a big window.
-
- 11 Reach for UDP when timeliness beats completeness, and accept that you then own reliability, ordering and congestion yourself.
-
- 12 Program the socket as a byte stream with partial reads and writes, and never block one connection on another, use an event loop at scale.

-
- 13 Read an address as network-plus-host under a prefix; subnet with CIDR, and learn the special ranges before you have to debug them.
-
- 14 Assume most hosts sit behind NAT with a private address, so inbound connections need deliberate help to arrive.
-
- 15 Remember it's always DNS: resolution is a cached hierarchy, and a wrong TTL or stale record outlives the change that caused it.
-
- 16 Optimise latency, not just bandwidth; distance is a floor set by the speed of light, so make fewer round trips over shorter ones.
-
- 17 Put copies close with a CDN and let anycast route and fail over, but treat that CDN as a global single point of failure.
-
- 18 Put a load balancer with real health checks in front of stateless backends, so any server serves any request and failover is free.
-
- 19 Localise a fault to one layer before you change anything, and start by asking where the time goes.
-
- 20 Know the whole journey from URL to pixels, and make it fast the only way there is: fewer round trips, over shorter distances.
-

READING LIST

The books, courses and specifications behind this volume, and the chapters each one deepens. If you read only one, make it *Kurose & Ross*; if you do only one thing, build a working TCP stack yourself in the CS 144 labs, so the transport layer stops being a black box, exactly the deliverable this stage of the syllabus asks you to ship.

WORK	AUTHOR	DEEPENS
Computer Networking: A Top-Down Approach	Kurose & Ross	Ch. 1–20
CS 144: Introduction to Computer Networking (labs)	Stanford	Ch. 9–12
High Performance Browser Networking	Ilya Grigorik	Ch. 3–6, 16–17
TCP/IP Illustrated, Vol. 1	Fall & Stevens	Ch. 9–14
Beej's Guide to Network Programming	Brian “Beej” Hall	Ch. 12
Computer Networks: A Systems Approach	Peterson & Davie	Ch. 1, 13–14
Bulletproof SSL and TLS	Ivan Ristić	Ch. 6
MDN Web Docs: HTTP	Mozilla	Ch. 2–7

Compiled October 2026 as the companion to stage 06, Networking and the web, of The Engineer's Syllabus, alongside *The Engineer's Craft*, *Shipyard*, *Fleet*, *Watch*, *Citadel* and *Foundry*. It is deliberately top-down, starting from HTTP and working down to the wire, the way Kurose & Ross teach it. Research drew on Kurose & Ross's *Top-Down Approach*, Ilya Grigorik's *High Performance Browser Networking*, Beej's Guide to Network Programming, the core specifications, RFC 9110/9112 (HTTP), RFC 9114 (HTTP/3) and RFC 9000 (QUIC), RFC 8446 (TLS 1.3) and RFC 6455 (WebSocket), MDN's HTTP documentation and web.dev on Core Web Vitals, Cloudflare's learning center and Radar for adoption figures, and the public post-mortems of the outages used as cautionary tales: the Facebook 2021 BGP/DNS withdrawal, the 2016 Dyn/Mirai DNS attack, the AWS us-east-1 DNS race of October 2025, Cloudflare's November 2025 config outage, the Fastly 2021 outage, and HTTP/2 Rapid Reset (CVE-2023-44487). Addresses, versions and adoption percentages are late-2026 figures used to teach method, not facts to rely on.

Every self-contained example was executed before publication, in pure-standard-library Python, so the book runs the same way every time. Re-checked in October 2026 with Python 3.14.7: every self-contained example ran and printed what the text says it prints. Tested here: packet encapsulation overhead; URL parsing, origins and query encoding; HTTP request and response framing with Content-Length and chunked decoding; the cache state machine (miss, hit, revalidate) with ETag and max-age; the head-of-line-blocking comparison across HTTP/1.1, 2 and 3; the certificate-chain verifier; the RFC 6455 WebSocket accept-key; protobuf wire encoding and schema evolution; the TCP cumulative-ACK and retransmission model; the congestion-control sawtooth and the bandwidth-delay product; the UDP-versus-TCP timeliness demo; a real loopback socket echo; IPv4 and IPv6 address and subnet maths with the standard library's `ipaddress`; the NAT translation table; the DNS resolution walk with caching; the latency budget and the distance-dominates demonstration; the anycast nearest-point-of-presence routing and failover; the layer-4-versus-layer-7 load balancer with health checks; the `curl` timing breakdown that localises a slow request to one layer; and the end-to-end round-trip accounting. Three captures use the live network and are labelled illustrative: the TLS handshake and certificate chain from `openssl s_client`, DNS records from `dig`, and a `curl` timing capture against example.com.