

Internet Networks

Aayush Bajaj

July 10, 2026

Contents

1	layered models	1
2	physical and link layer	2
3	the network layer: IP	2
3.1	addressing and CIDR	3
3.2	forwarding: longest-prefix match	3
3.3	NAT	3
4	transport: UDP and TCP	4
4.1	UDP — the honest datagram	4
4.2	TCP — the reliable bytestream	4
4.3	the three-way handshake	4
4.4	flow control: the sliding window	5
4.5	congestion control: probing a network that won't tell you anything	5
5	DNS: one resolution, end to end	6
6	HTTP: 1.1, 2, 3	6
7	TLS in one sketch	7
8	the sockets API	7
9	see also	8

the internet is a triumph of indirection: no layer trusts the one below to be reliable, timely, or even present, and yet a packet leaves your laptop, crosses a dozen autonomous systems owned by companies that actively dislike each other, and arrives. this page walks the stack bottom-up, then follows one HTTP request through DNS, TCP, and TLS to see every layer earn its keep.

*the design bet — smart edges,
dumb core — is called the
end-to-end principle*

1 layered models

two models, one running joke: OSI is the model everyone teaches and nobody implements; TCP/IP is the model everyone implements and nobody can cleanly draw. layers 5–7 of OSI collapse into “application” in practice (Kurose, James F. and Ross, Keith W., 2020).

OSI layer	responsibility	TCP/IP layer	example protocols
7 application	user-facing semantics	application	HTTP, DNS, SMTP, SSH
6 presentation	encoding, encryption, compression	application	TLS(ish), MIME
5 session	dialogue management, checkpoints	application	(mostly folklore)
4 transport	process-to-process delivery, reliability	transport	TCP, UDP, QUIC
3 network	host-to-host delivery across networks	internet	IP, ICMP, BGP, OSPF
2 data link	frame delivery across one hop	link	ethernet, wi-fi, ARP
1 physical	bits on a medium	link	twisted pair, fibre, RF

each layer treats the layer above’s entire message as opaque payload and prepends its own header — *encapsulation*. the receiving stack peels the headers off in reverse.

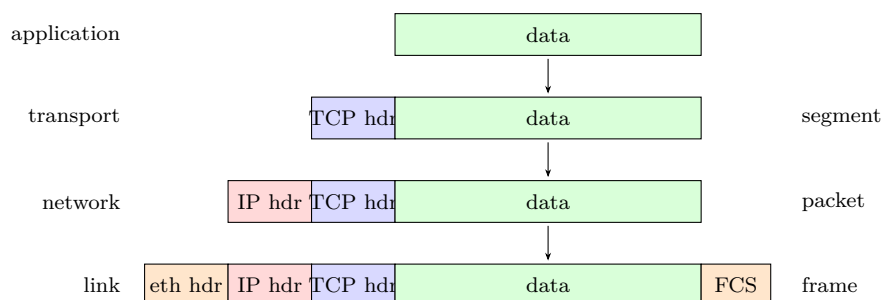


Figure 1: encapsulation: each layer wraps the one above. the wire carries the whole onion.

2 physical and link layer

the link layer’s job: get a frame across *one hop* — one cable, one wi-fi cell, one switch fabric.

- **ethernet framing.** an ethernet II frame: 6-byte destination MAC, 6-byte source MAC, 2-byte ethertype (0x0800 for IPv4, 0x0806 for ARP), 46–1500 bytes of payload, 4-byte CRC-32 frame check sequence. the wire also carries a 7-byte preamble + 1-byte start delimiter for clock sync, not counted in the 64-byte minimum frame size.
- **MAC addresses** are 48-bit, flat (no hierarchy — you cannot route on them), assigned per interface. they only ever matter on the local segment: the frame’s MACs are rewritten at every router hop, while the IP addresses inside stay put.
- **ARP** bridges layers 2 and 3: to send an IP packet to a host on your subnet you broadcast “who has 192.168.5.201?” and the owner replies unicast with its MAC. results are cached (and cache poisoning is a classic LAN attack).
- **hubs vs switches.** a hub is layer 1: it repeats every bit out every port — one collision domain, everyone shares the bandwidth. a switch is layer 2: it learns which MACs live behind which ports from the source addresses of frames it forwards, then sends each frame only where it needs to go — per-port collision domains, full duplex, and the reason hubs are extinct (Kurose, James F. and Ross, Keith W., 2020).

the 1500-byte payload cap is the MTU — the number behind every fragmentation headache

3 the network layer: IP

IP delivers a datagram from any host to any host, best-effort: no ordering, no retransmission, no guarantee at all. everything reliable is built above it.

3.1 addressing and CIDR

an IPv4 address is 32 bits. CIDR notation **a.b.c.d/n** says the top n bits are the network prefix, the remaining $32 - n$ bits name hosts. a **/26** therefore holds $2^{32-26} = 64$ addresses, of which 62 are usable (all-zeros names the network, all-ones is broadcast).

worked example — carve **192.168.5.0/24** into four equal subnets. we need 2 extra prefix bits, giving **/26** blocks of 64:

subnet	netmask	usable hosts	broadcast
192.168.5.0/26	255.255.255.192	192.168.5.1 – 192.168.5.62	192.168.5.63
192.168.5.64/26	255.255.255.192	192.168.5.65 – 192.168.5.126	192.168.5.127
192.168.5.128/26	255.255.255.192	192.168.5.129 – 192.168.5.190	192.168.5.191
192.168.5.192/26	255.255.255.192	192.168.5.193 – 192.168.5.254	192.168.5.255

to place **192.168.5.201**: the last octet is $201 = 11001001_2$; its top two bits **11** select the fourth block, so it lives in **192.168.5.192/26**. equivalently $201 \wedge 192 = 192$ — AND the address with the mask, read off the network.

*sanity-checked with python's
ipaddress module; subnetting by
hand is a rite of passage but
machines don't fumble octets*

3.2 forwarding: longest-prefix match

a router's forwarding table maps prefixes to next hops. the rule: among all prefixes that match the destination, the *longest* (most specific) wins.

prefix	next hop
0.0.0.0/0	isp uplink
10.0.0.0/8	core router
10.1.0.0/16	building B
10.1.2.0/24	lab switch

destination **10.1.2.3** matches all four rows; **10.1.2.0/24** is longest, so the packet goes to the lab switch. destination **10.9.9.9** matches only the **/8** and the default, so it goes to the core router. this one rule is why an operator can advertise a broad prefix and carve exceptions out of it — the whole BGP ecosystem leans on it.

3.3 NAT

IPv4 ran out of addresses, so your router lies. network address translation rewrites outbound packets' source (**private ip, port**) to (**public ip, fresh port**) and keeps the mapping in a table; inbound replies are translated back. consequences worth knowing:

- an entire household presents as one address; the port number is doing the demultiplexing that addresses were meant to do.
- unsolicited inbound connections have no table entry and die — an accidental firewall, and deliberate pain for p2p and game servers (hence port forwarding, STUN, and hole punching).
- NAT boxes must peek into layer 4 to grab ports, cheerfully violating layering. purists grumble; the internet shipped (Kurose, James F. and Ross, Keith W., 2020).

4 transport: UDP and TCP

4.1 UDP — the honest datagram

UDP’s entire header is 8 bytes: source port, destination port, length, checksum. it adds exactly one thing to IP: ports, i.e. process-level addressing. no ordering, no reliability, no connection state. that austerity is a feature — DNS, game state, streaming, and QUIC all build their own semantics on top.

4.2 TCP — the reliable bytestream

the header fields that actually matter:

- **sequence number** (32-bit): the byte offset of this segment’s first byte within the stream. TCP numbers *bytes*, not segments.
- **acknowledgement number**: the next byte the receiver expects — a cumulative ACK.
- **flags**: SYN (synchronise sequence numbers), ACK, FIN (polite close), RST (impolite close).
- **window**: how many more bytes the receiver is willing to buffer — flow control, advertised on every segment.
- **options**: MSS, window scaling, SACK, timestamps — a 1981 header wearing 2020s prosthetics.

4.3 the three-way handshake

both sides must learn each other’s initial sequence number before any byte can be acknowledged, and a two-message exchange cannot confirm both directions — hence three.

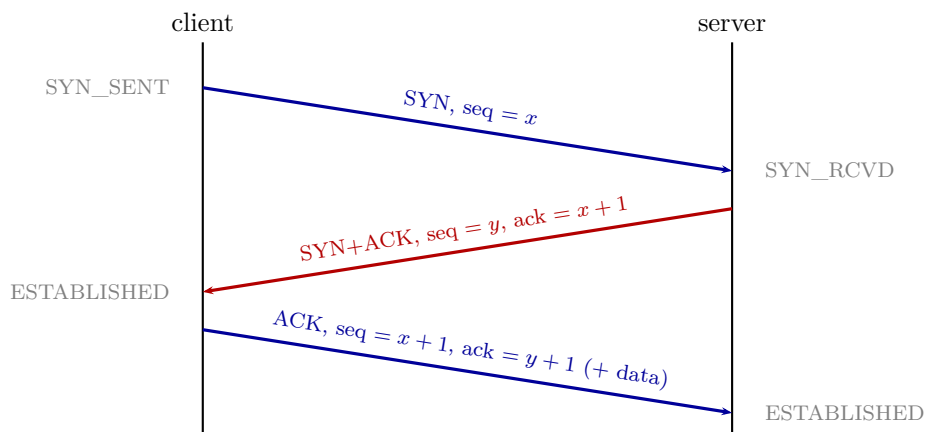


Figure 2: three-way handshake. SYN and SYN+ACK each consume one sequence number; the final ACK may already carry data.

teardown is symmetric but four-way (FIN, ACK, FIN, ACK), and the side that closes first lingers in `TIME_WAIT` for two maximum segment lifetimes — the reason a restarted server whines “address already in use” until you set `SO_REUSEADDR`.

4.4 flow control: the sliding window

the receiver advertises a window `rwnd` — how much buffer it has left. the sender may have at most `rwnd` unacknowledged bytes in flight. as ACKs arrive, the window's left edge slides forward and frees budget for new bytes. throughput is therefore capped at

$$\text{throughput} \leq \frac{\text{rwnd}}{\text{RTT}}, \quad (1)$$

which is why the original 16-bit (64 KiB) window strangled fast long-distance links and window scaling had to be bolted on: a 100 ms transpacific path needs `rwnd` $\approx$ 12.5 MB to fill a 1 Gbps pipe.

4.5 congestion control: probing a network that won't tell you anything

flow control protects the receiver; congestion control protects the network. the sender maintains a second cap, `cwnd`, and keeps at most $\min(\text{rwnd}, \text{cwnd})$ bytes in flight. classic TCP (reno) infers congestion purely from loss (Kurose, James F. and Ross, Keith W., 2020):

- **slow start**: `cwnd` starts at a few MSS and grows by one MSS per ACK received — doubling every RTT. exponential, despite the name.
- **congestion avoidance**: past the threshold `sssthresh`, growth slows to about one MSS per RTT — additive increase.
- **fast retransmit / fast recovery**: three duplicate ACKs mean one segment died but data is still flowing; retransmit immediately, set `sssthresh` = `cwnd`/2, and resume additive increase from there — multiplicative decrease.
- **timeout**: silence means things are genuinely bad — drop `cwnd` to 1 MSS and slow-start again.

additive increase + multiplicative decrease (AIMD) traces the famous sawtooth, and it is the AIMD combination specifically that drives competing flows toward an equal share of the bottleneck.

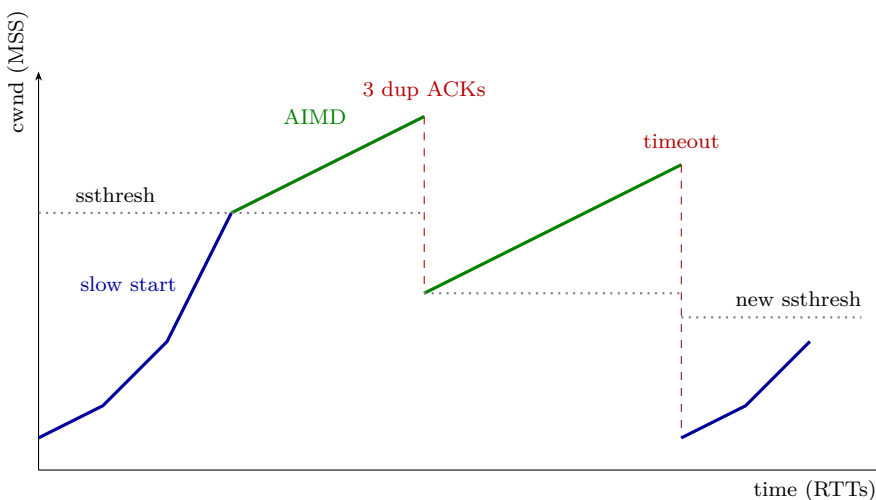


Figure 3: the reno sawtooth: exponential slow start, additive increase, multiplicative decrease on triple-duplicate ACK, collapse to 1 MSS on timeout.

modern stacks have moved on — cubic grows the window as a cubic function of time since the last loss, and BBR models bottleneck bandwidth and RTT directly instead of waiting for loss — but the sawtooth is still the mental model everything else is measured against.

loss-based control ages badly on wireless links, where loss often means interference, not congestion

5 DNS: one resolution, end to end

you type `abaj.ai`. what happens, assuming every cache is cold:

- **stub resolver** (your OS) sends a UDP query to its configured recursive resolver, say `1.1.1.1`.
- **recursive resolver** asks a **root server**: “A record for `abaj.ai`?” the root doesn’t know, but replies with the nameservers for the `.ai` top-level domain — a referral.
- it asks a **TLD server** for `.ai`, which refers it onward to the **authoritative nameservers** for `abaj.ai`.
- it asks the **authoritative server**, which finally answers with the A record and a TTL.
- the answer is cached at every level for TTL seconds and handed back to the stub. subsequent lookups from anyone behind that resolver are one hop.

the hierarchy means no single server knows the whole namespace, and caching means the roots survive despite the entire planet resolving names constantly. record types worth knowing: **A** and **AAAA** (IPv4/IPv6 address), **CNAME** (alias), **MX** (mail), **NS** (delegation), **TXT** (structured graffiti: SPF, domain verification) (Kurose, James F. and Ross, Keith W., 2020).

6 HTTP: 1.1, 2, 3

HTTP/1.1 (1997) is plain text over TCP: one request, then one response, framed by **Content-Length** or chunked encoding. persistent connections and pipelining were added, but responses must return in order, so one slow response blocks every request queued behind it — application-level head-of-line blocking. browsers coped by opening several parallel connections per origin, a workaround wearing a standard’s clothes.

HTTP/2 (2015) keeps HTTP semantics but replaces the text framing with binary frames multiplexed over *one* TCP connection: many concurrent streams, per-stream flow control, HPACK header compression. this fixes head-of-line blocking at the HTTP layer — but not at the transport. lose one TCP segment and every stream stalls behind it, because TCP promises a single ordered bytestream and keeps that promise stubbornly.

HTTP/3 (2022) gives up on TCP and runs over QUIC on UDP. QUIC provides multiple independently-ordered streams, so a lost packet stalls only the stream it belonged to; TLS 1.3 is fused into the transport handshake (one round trip to first byte, zero for resumed sessions); and connections are identified by a connection ID rather than the address 4-tuple, surviving a phone hopping from wi-fi to cellular. the price: QUIC lives in userspace, so every application ships its own transport — which is precisely why it can evolve faster than TCP ever could (Kurose, James F. and Ross, Keith W., 2020).

7 TLS in one sketch

TLS 1.3 needs a single round trip:

- **client hello**: supported cipher suites, plus an ephemeral diffie–hellman key share, sent optimistically.
- **server hello**: the chosen suite and the server’s own key share — both sides can now derive the shared secret, and *everything after this point is already encrypted*.
- **certificate + certificate-verify**: the server presents its certificate chain and signs the handshake transcript with the certificate’s private key, proving it isn’t a bystander replaying bytes.
- **finished** (both directions): MACs over the whole transcript confirm nobody tampered with the negotiation.

authentication comes from the certificate chain ending at a root your OS already trusts; confidentiality from the ephemeral diffie–hellman secret. because those keys are ephemeral, recording traffic today and stealing the certificate key tomorrow decrypts nothing — forward secrecy.

8 the sockets API

the kernel exposes all of the above through half a dozen syscalls; the sequence is a little dance both sides must know (Tanenbaum, Andrew S., 2008):

server	client
<code>socket()</code>	<code>socket()</code>
<code>bind() addr:port</code>	
<code>listen() backlog</code>	
<code>accept() ... blocks</code>	<code>connect()</code> (handshake happens here)
<code>recv() / send()</code>	<code>send() / recv()</code>
<code>close()</code>	<code>close()</code> (FIN)

`accept()` returns a *new* file descriptor per connection — the listening socket keeps listening. an echo pair, run for real:

```
1 # echo_server.py
2 import socket
3
4 srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # socket()
5 srv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
6 srv.bind(("127.0.0.1", 5555)) # bind()
7 srv.listen(1) # listen()
8 print("server: listening on 127.0.0.1:5555")
9
10 conn, addr = srv.accept() # accept() blocks
11 print(f"server: connection from {addr[0]}:{addr[1]}")
12 while True:
13     data = conn.recv(1024) # recv()
14     if not data: # peer sent FIN
15         break
16     print(f"server: echoing {data!r}")
17     conn.sendall(data) # send()
18 conn.close()
19 srv.close()
20 print("server: done")
```

```

1 # echo_client.py
2 import socket
3
4 cli = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # socket()
5 cli.connect(("127.0.0.1", 5555)) # connect() ->
6 ↪ handshake
7 for msg in (b"hello", b"three-way handshake complete"):
8     cli.sendall(msg) # send()
9     reply = cli.recv(1024) # recv()
10    print(f"client: sent {msg!r}, got {reply!r}")
11 cli.close() # close() -> FIN

```

server started first, then the client; interleaved output from the two processes:

```

client: sent b'hello', got b'hello'
client: sent b'three-way handshake complete', got b'three-way handshake complete'
--- server output ---
server: listening on 127.0.0.1:5555
server: connection from 127.0.0.1:52661
server: echoing b'hello'
server: echoing b'three-way handshake complete'
server: done

```

one honesty note: `recv()` returns *whatever bytes have arrived*, not “one message” — TCP has no message boundaries. this toy works because each send fits in one segment and the two sides alternate strictly. real protocols add their own framing (length prefixes, delimiters), which is exactly what i had to do in [the LAN messenger](#) — a threaded TCP+UDP chat written in C, where the framing (and the `SO_REUSEADDR` lesson above) were learned the hard way.

9 see also

- [LAN messenger](#) — this whole page, compiled into a C project
- [linux](#) — the kernel that owns the sockets
- [databases](#) — what usually answers on port 5432
- [memory](#) — where all those buffers live

references

Kurose, James F. and Ross, Keith W. (2020). *Computer Networking: A Top-Down Approach*, Pearson.

Tanenbaum, Andrew S. (2008). *Modern Operating Systems*, Pearson.