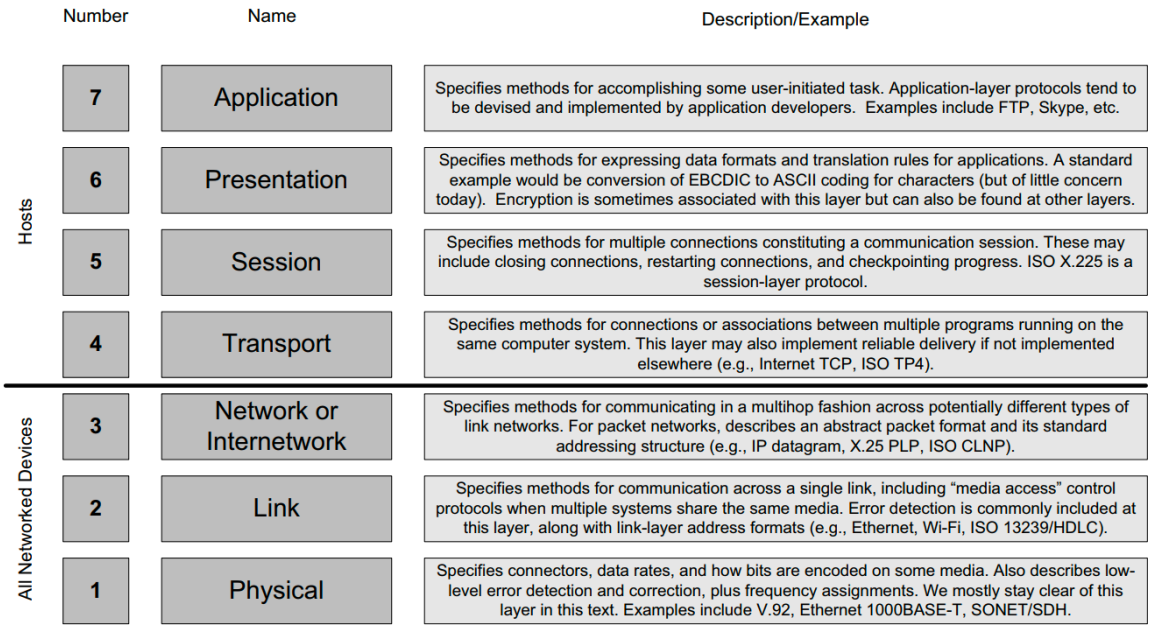
NETWORKING

General Networking Questions

- *Can you explain what is the OSI model, its different layers with a brief explanation of each?*
  - The OSI model is a conceptual framework that breaks end-to-end communication into seven stacked layers, each with a specific role and clean interfaces to the layers above and below. It's mainly a teaching and reasoning tool—real networks (like the Internet) don't implement it literally—but it's very useful vocabulary.
  - At the **Physical layer (Layer 1)**, bits are turned into signals on a medium and back again. This is where voltages, light levels, modulation, line coding, connectors, and link speed live. Ethernet PHYs, fiber optics, copper pairs, and radio are all Layer-1 concerns; there are no packets here, just timing and symbols.
  - The **Data Link layer (Layer 2)** includes those protocols and methods for establishing connectivity to a neighbor sharing the same medium. Some link-layer networks (e.g., DSL) connect only two neighbors. When more than one neighbor can access the same shared network, the network is said to be a multi-access network. Wi-Fi and Ethernet are examples of such multi-access link-layer networks, and specific protocols are used to mediate which stations have access to the shared medium at any given time. .
  - At the **Network layer (Layer 3)**, packets are routed across multiple links and networks. IP (v4/v6) is the canonical example: it assigns logical addresses, fragments if needed, and forwards packets through routers according to a routing table and protocols like OSPF or BGP. Layer-3 is where end-to-end reachability and path selection happen.
  - At the **Transport layer (Layer 4)**, processes get end-to-end conversations with specific delivery semantics. TCP provides a reliable, ordered byte stream with flow and congestion control; UDP provides connectionless, message-oriented delivery with no built-in reliability. Port numbers, sockets, and the notion of "a connection" or "a datagram" are Layer-4 concepts.
  - At the **Session layer (Layer 5)**, long-lived dialogs are established, managed, and torn down, historically handling things like checkpoints and recovery across interruptions. In modern Internet practice, explicit "session" functions are usually folded into higher-level protocols or the transport itself; still, it's useful to label "who's logged in and how we resume" as session semantics.
  - At the **Presentation layer (Layer 6)**, data is transformed so both sides agree on representation. This includes serialization, encoding, compression, and encryption. TLS handshakes and record protection, ASN.1, JSON, and Protobuf encoding are all presentation concerns: same meaning, agreed format, independent of transport.

- At the **Application layer (Layer 7)**, the actual application protocols and semantics live. HTTP, DNS, SMTP, FIX, WebSockets, and exchange market-data formats are Layer-7: methods, messages, resources, and business rules. Everything below exists to carry this layer's intent.
- It helps to map this to the pragmatic "TCP/IP" model: Physical and Data Link correspond to the Link layer, IP is the Internet layer, TCP/UDP are the Transport layer, and everything else—session, presentation, application—collapses into "Application." The OSI stack's value in interviews is the shared mental model: when you discuss a bug or design, you can pinpoint whether it's a Layer-2 MAC learning issue, a Layer-3 routing problem, a Layer-4 congestion/backpressure effect, or a Layer-7 protocol/encoding mistake.



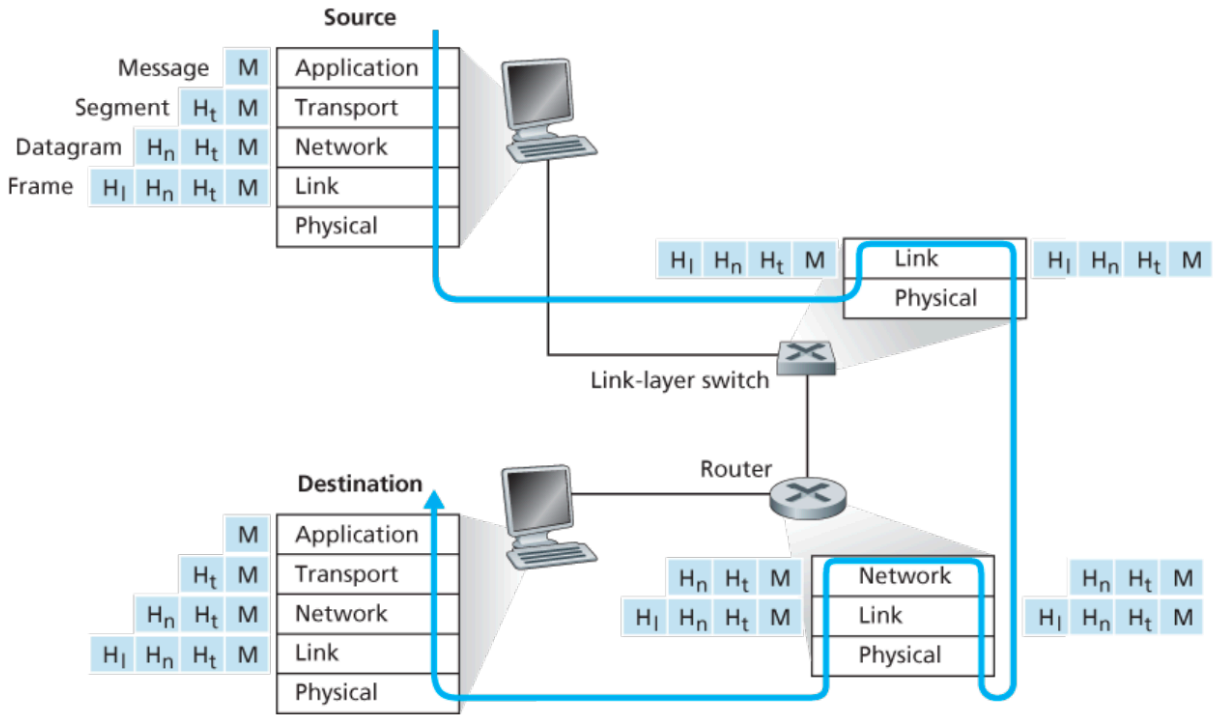
	Number	Name	Description/Example
Hosts	7	Application	Specifies methods for accomplishing some user-initiated task. Application-layer protocols tend to be devised and implemented by application developers. Examples include FTP, Skype, etc.
	6	Presentation	Specifies methods for expressing data formats and translation rules for applications. A standard example would be conversion of EBCDIC to ASCII coding for characters (but of little concern today). Encryption is sometimes associated with this layer but can also be found at other layers.
	5	Session	Specifies methods for multiple connections constituting a communication session. These may include closing connections, restarting connections, and checkpointing progress. ISO X.225 is a session-layer protocol.
	4	Transport	Specifies methods for connections or associations between multiple programs running on the same computer system. This layer may also implement reliable delivery if not implemented elsewhere (e.g., Internet TCP, ISO TP4).
All Networked Devices	3	Network or Internetwork	Specifies methods for communicating in a multihop fashion across potentially different types of link networks. For packet networks, describes an abstract packet format and its standard addressing structure (e.g., IP datagram, X.25 PLP, ISO CLNP).
	2	Link	Specifies methods for communication across a single link, including "media access" control protocols when multiple systems share the same media. Error detection is commonly included at this layer, along with link-layer address formats (e.g., Ethernet, Wi-Fi, ISO 13239/HDLC).
	1	Physical	Specifies connectors, data rates, and how bits are encoded on some media. Also describes low-level error detection and correction, plus frequency assignments. We mostly stay clear of this layer in this text. Examples include V.92, Ethernet 1000BASE-T, SONET/SDH.

- To be clear:
  - **L7–L5** make the message.
  - **L4 (TCP/UDP)** adds a **transport header** (ports, sequence, etc.) → a **segment/datagram**.
  - **L3 (IP)** wraps that whole L4 unit with an **IP header** (source/dest IP, TTL, etc.) → a **packet**.
  - **L2 (Ethernet)** wraps the L3 packet with a **frame header/trailer** (MACs, FCS) → a **frame**.
  - **L1** puts the bits on the wire.
- See this video for more info.

- *How do each layer intricates with each other?*

1. At the sending host, an **application-layer message** (M in Figure 1.24) is passed to the transport layer. In the simplest case, the transport layer takes the message and appends additional information (so-called **transport-layer header information**, H in Figure 1.24) that will be used by the receiver-side transport layer. The application-layer message and the transport-layer header information together constitute the **transport layer segment**. The transport-layer segment thus encapsulates the application-layer message. The added information might include information allowing the receiver-side transport layer to deliver the message up to the appropriate application, and error-detection bits that allow the receiver to determine whether bits in the message have been changed in route.
2. The transport layer then passes the segment to the network layer, which adds **network-layer header information** (H in Figure 1.24) such as source and destination end system addresses, creating a **network-layer datagram** (sometimes also called a **network packet**).
3. The datagram is then passed to the link layer, which (of course!) will add its own **link-layer header information** and create a **link-layer frame**. Thus, we see that at each layer, a packet has two types of fields: header fields and a payload field. The payload is typically a packet from the layer above.

Figure 1.24 shows the physical path that data takes down a sending end system's protocol stack, up and down the protocol stacks of an intervening link-layer switch



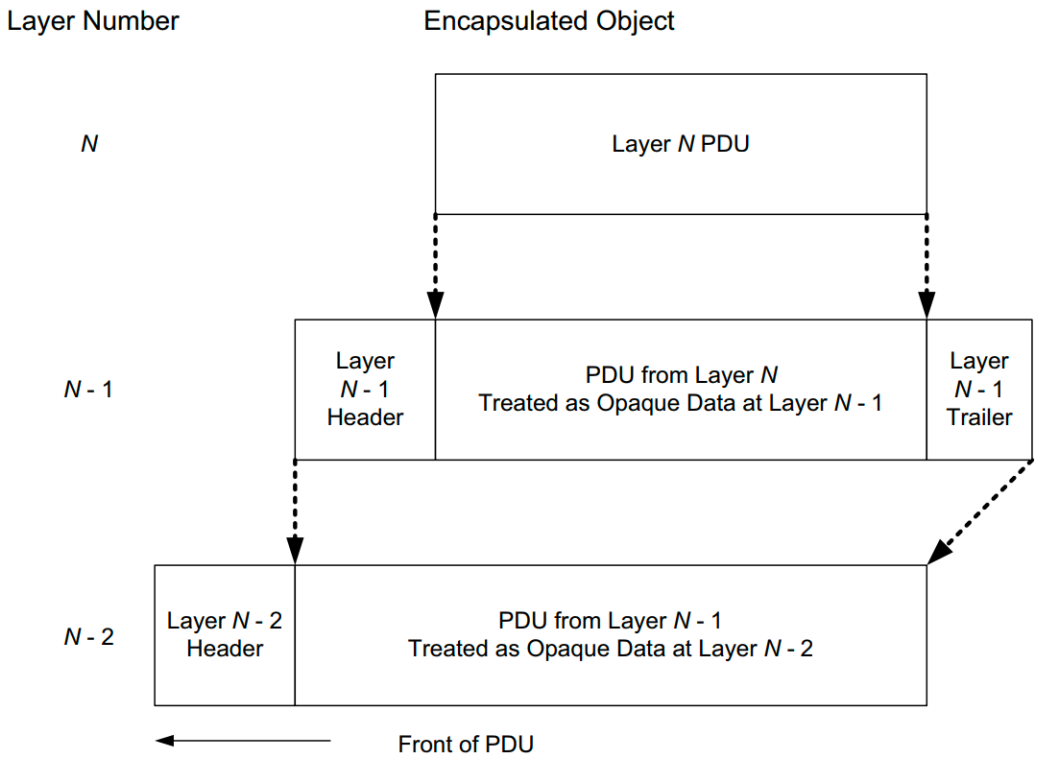


Figure 1-3 Encapsulation is usually used in conjunction with layering. Pure encapsulation involves taking the PDU of one layer and treating it as opaque (uninterpreted) data at the layer below. Encapsulation takes place at each sender, and decapsulation (the reverse operation) takes place at each receiver. Most protocols use headers during encapsulation; a few also use trailers.

- *Which layer in the TCP/IP protocol typically contains some error control?*
  - You can think of “error control” as being spread across several layers, each with its own scope. I’ll walk through the OSI layers from bottom to top and say what kind of error detection/correction you typically see at each one.
  - At the **Physical layer**, the job is to move raw bits as signals. Error control here is usually *built into the coding of the bits themselves*. Examples are things like parity bits, line codes with embedded redundancy, or forward error correction (FEC) on fiber and wireless links. These schemes can sometimes correct a few flipped bits without needing a retransmit, but they’re very local: they only cover the single hop on that physical medium.
  - At the **Data Link layer**, you get the first clear, explicit error detection that most people learn about. Frames almost always carry a checksum or a CRC (cyclic redundancy check) in a trailer field (e.g. Ethernet FCS). The receiver recomputes the CRC over the received frame and compares it. If it doesn’t match, the frame is discarded. Some data-link protocols go further and also do retransmissions on this hop (for example, Wi-Fi with ARQ), but others like basic Ethernet just drop the bad frame and rely on upper layers to recover.

- At the **Network layer**, the classic example is IP. IPv4 has a header checksum that protects only the header fields, not the payload. This helps catch corruption that might cause misrouting or confusion about packet length, but it doesn't guarantee payload correctness. IPv6 dropped even that checksum, on the assumption that underlying links already provide strong error detection and that transport protocols (like TCP) have end-to-end checksums. So at the network layer, you usually have little or no payload error control, and sometimes no checksum at all.
- At the **Transport layer**, you get full end-to-end error control. TCP segments carry a checksum over the header and payload; the receiver drops any segment with a bad checksum and relies on acknowledgments, timeouts, and retransmission to recover. This gives you a reliable byte stream between endpoints, even though the network underneath can drop or corrupt packets. UDP also has an optional checksum in IPv4 and a mandatory one in IPv6; it doesn't retransmit, but it can detect corrupted datagrams and discard them.
- Finally, at the **Application layer**, many protocols and applications add their own integrity checks on top of what TCP/UDP provide. For example, file transfer protocols may include hashes (MD5/SHA) of files, HTTP downloads are often verified against a checksum, and security protocols like TLS wrap application data in MACs or AEAD tags that detect any corruption or tampering, not just random bit flips. This is not about fixing physical noise on links, but about ensuring the data the application sees is exactly what was intended, with strong end-to-end integrity guarantees.

- *What is an IP packet?*
  An IP packet is the basic unit of data transmission at the network layer of the Internet protocol suite. It encapsulates data along with header information that contains essential routing and control details, such as the source and destination IP addresses, protocol version (IPv4 or IPv6), Time to Live (TTL), and various flags. The header ensures that the packet can be properly routed across network boundaries and delivered to the correct destination, while the payload carries the actual data, which might include a TCP segment, UDP datagram, or other protocol-specific information. Essentially, IP packets enable different networks to communicate with one another by providing the necessary structure and addressing information for data delivery over the Internet.

- *What is payload?*
  The payload is the portion of data transmitted in a packet that represents the actual intended information, as opposed to the protocol overhead such as headers or metadata required for routing and control. In network communications, for example, the payload might

be the contents of a web page or a file being transferred, while the surrounding headers contain details like source/destination addresses, checksums, and protocol-specific flags needed to ensure proper delivery and handling.

- *What is the difference between ad IP address and Ethernet address?*
  - An **IP address** is a **Layer-3 (network layer)** *logical* address that identifies an endpoint in the IP network so packets can be **routed hop-by-hop across multiple networks** (e.g., IPv4 = 32 bits like 203.0.113.7; IPv6 = 128 bits). It's configured/assigned (DHCP, static), can change with location or interface, and stays the same from the sender's view as the packet traverses routers (NAT aside).
  - An **Ethernet address** (MAC) is a **Layer-2 (data link)** *physical/link-local* address that identifies a **NIC on a single LAN segment** (typically 48 bits like 00:1A:2B:3C:4D:5E). Switches use MACs to forward **frames** on the local link; MAC headers are **rewritten at every hop** by routers, while the IP header persists end-to-end. ARP (for IPv4) and Neighbor Discovery (for IPv6) map an IP address to a MAC so a host can send the next-hop frame.
  - In short: **IP = routable, logical, end-to-end identity; MAC = link-local, hop-to-hop delivery on Ethernet.**

- *What is 'switching' in the context of networking? What is the difference between packet-switching and circuit-switching?*
  - Switching is how a network's intermediate devices decide **where to send traffic next** so data can travel hop-by-hop from a source to a destination. The two classic paradigms are **circuit switching** and **packet switching**; they differ in how resources are allocated and how data is carried.
  - In **circuit switching**, the network first establishes an end-to-end **circuit**—a reserved slice of link capacity along a fixed path—using TDM/FDM or equivalent reservation. While the circuit is up, the sender has a guaranteed rate and the path is exclusive, so once transmission begins there's essentially **no queueing** in the network and delay is stable. You pay a **setup time** before sending, and bandwidth can be **wasted during silence** because it stays reserved whether or not you have data. Classic telephony (PSTN) is the textbook example; modern analogs include leased lines or strict QoS reservations.
  - In **packet switching**, the sender breaks data into **packets**; each packet is forwarded independently at every hop based on its header. Links are **statistically multiplexed**: many flows share them, taking turns as packets arrive. This yields **high utilization and robustness** (no advance reservation, rerouting around failures, easy scalability),

but introduces **variable delay** because packets can queue behind others and may arrive out of order or be dropped under congestion. The Internet is a packet-switched network. Within packet switching you'll see (a) **datagram mode** (pure IP: no setup, each packet self-routed) and (b) **virtual-circuit mode** (e.g., ATM, Frame Relay, MPLS-TE) where a lightweight setup establishes a label/path but still shares links statistically.

- Circuit switching pre-allocates use of the transmission link regardless of demand, with allocated but unneeded link time going unused. Packet switching on the other hand allocates link use on demand. Link transmission capacity will be shared on a packet-by-packet basis only among those users who have packets that need to be transmitted over the link
- Trade-offs are straightforward: circuit switching gives **predictable latency and bandwidth guarantees** at the cost of setup time and potential underutilization; packet switching gives **efficiency, flexibility, and fault tolerance** at the cost of **queueing jitter, possible loss, and reordering**. Many real systems mix the ideas—for example, MPLS traffic engineering or data-center QoS can approximate circuit-like guarantees atop a packet-switched core.

- *What do exchanges use with trading firms? Packet or circuit switching?*
  - Short answer: **packet switching.**
  - Exchanges distribute market data to firms as **IP packets**—typically **UDP multicast** for feeds (e.g., ITCH/MDP variants) and **TCP** for order entry/control (e.g., FIX, binary gateways). In colocation you get an **Ethernet cross-connect** to the exchange's network; over metro/long-haul you might lease a "private circuit" or MPLS link, but that's just a provisioning term—the payload is still **packet-switched IP/Ethernet**.
  - So while firms may buy dedicated connectivity (for isolation and SLAs), the traffic itself is not classic circuit-switched like PSTN; it's packets, often with redundancy (A/B multicast), QoS, and sometimes engineered for determinism, but still **packet-switched networking end to end**.

- *Can you explain what is 'multiplexing'?*
  - Multiplexing = sharing one channel among many independent streams by **combining them in a controlled way** at the sender (mux) and **separating them** at the receiver (demux). It raises utilization of a scarce resource (wire, radio band, CPU, thread) at the cost of coordination and sometimes extra delay.
  - Common forms:

    - **TDM (Time-Division):** each stream gets time slots on the same link (classic digital telephony).
    - **FDM/WDM (Frequency/Wavelength-Division):** each stream uses a distinct frequency/wavelength simultaneously (radio, fiber optics).
    - **CDM (Code-Division):** streams share time+frequency but use orthogonal codes; receiver correlates to recover each one.
    - **Statistical multiplexing (packets):** Internet style—packets from many flows queue on the same link and go whenever the link is free; highly efficient but adds queueing jitter/loss under load.
- Related host concepts:
    - **Link/flow multiplexing:** many flows share one NIC/link; demux by headers (IP addresses, ports).
    - **I/O multiplexing:** one thread waits on many sockets and services whichever is ready (`select/poll/epoll`).
    - **CPU scheduling:** the OS time-multiplexes cores among threads.
- Key trade-off: circuit-like schemes (TDM/FDM/CDM) give predictable shares; packet/statistical multiplexing gives flexibility and efficiency but variable delay.
- See this video for more info.

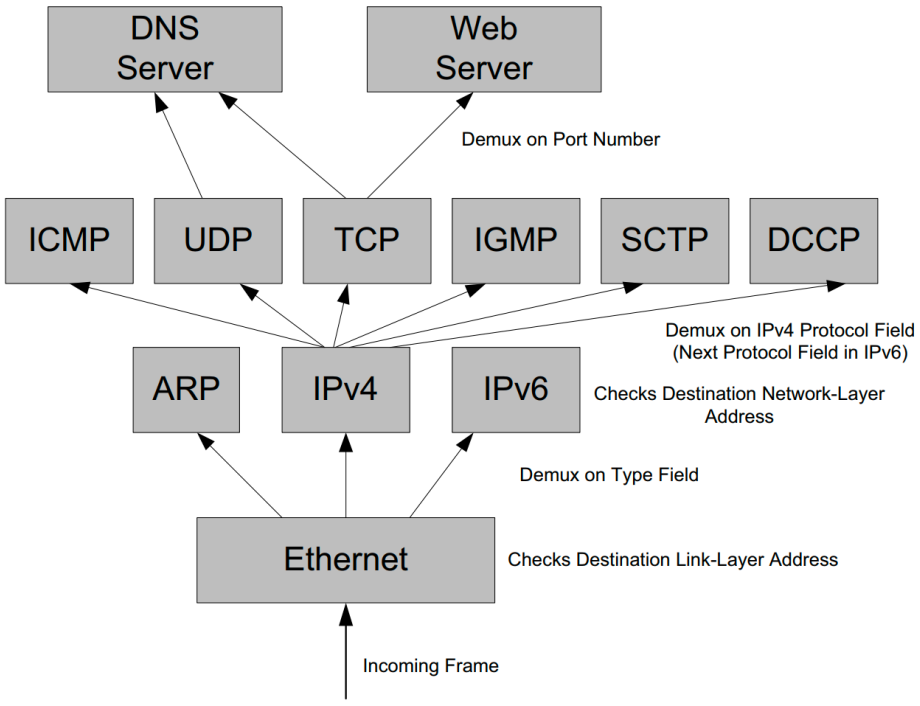


Figure 1-6 The TCP/IP stack uses a combination of addressing information and protocol demultiplexing identifiers to determine if a datagram has been received correctly and, if so, what entity should process it. Several layers also check numeric values (e.g., checksums) to ensure that the contents have not been damaged in transit.

- *A 1,280,000-bit file must be sent from Host A to Host B over a circuit-switched network. Every link uses TDM with 32 slots and has a raw bit rate of 2.048 Mbps. Establishing the end-to-end circuit takes 300 ms. Ignoring propagation delay, how long does it take to send the file?*
  - Each circuit gets a rate of $2.048\ \text{Mbps}/32 = 64\ \text{kbps}$.
  - The transmission time is $1{,}280{,}000\ \text{bits}/64{,}000\ \text{bps} = 20\ \text{s}$.
  - Add the circuit setup time: $20\ \text{s} + 0.3\ \text{s} = \mathbf{20.3}\ \text{s}$.

- *How long does it take a packet of length 1,000 bytes to propagate over a link of distance 2,500 km, propagation speed m/s, and transmission rate 2 Mbps? More generally, how long does it take a packet of length L to propagate over a link of distance d, propagation speed s, and transmission rate R bps? Does this delay depend on packet length? Does this delay depend on transmission rate?*
  - "Propagate" means the time for the signal to travel the link, i.e., propagation delay $d_{\text{prop}} = d/s$.
  - With $d = 2{,}500\,\text{km} = 2.5 \times 10^6\,\text{m}$ and $s = 2.5 \times 10^8\,\text{m/s}$ (typical for fiber), $d_{\text{prop}} = 2.5 \times 10^6 / 2.5 \times 10^8 = 0.01\,\text{s} = 10\,\text{ms}$. It does **not** depend on packet length or transmission rate.
  - If you instead want the "time until the last bit arrives" (ignoring queueing/processing), add transmission delay $d_{\text{trans}} = L/R$. For $L = 1000$ bytes $= 8000$ bits and $R = 2\,\text{Mb/s}$, $d_{\text{trans}} = 8000/2 \times 10^6 = 0.004\,\text{s} = 4\,\text{ms}$, so end-to-end = $10\,\text{ms} + 4\,\text{ms} = 14\,\text{ms}$.
  - In general: propagation $= d/s$ (independent of L and R); transmission $= L/R$; total (no queueing/processing) $= d/s + L/R$

- *Suppose Host A wants to send a large file to Host B. The path from Host A to Host B has three links, of rates a. Assuming no other traffic in the network, what is the throughput for the file transfer? b. Suppose the file is 4 million bytes. Dividing the file size by the throughput, roughly how long will it take to transfer the file to Host B?*
  - The end-to-end throughput over a path is the **bottleneck link rate**:

$$\text{throughput} = \min\{R_1, R_2, R_3\}.$$

  - For a 4-million-byte file $= 32{,}000{,}000$ bits, the transfer time is

$$T = \frac{\text{file size (bits)}}{\min\{R_1, R_2, R_3\}}.$$

  - Numeric example (common textbook rates)

If $R_1 = 500\ \text{kbps},\ R_2 = 2\ \text{Mbps},\ R_3 = 1\ \text{Mbps}$, then throughput $= \min = 500\ \text{kbps}$ kbps and

$$T = \frac{32 \times 10^6\ \text{bits}}{500 \times 10^3\ \text{bits/s}} = 64\ \text{s}.$$

- *Can you give the 4 different types of delays that compose a nodal delay?*
  - The total **nodal delay** is the sum of four components:

$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}.$$

  - **Processing delay** d_{proc} is the time a router/host spends examining the packet header, verifying checksums, updating tables, and deciding the outbound link; it's typically microseconds but can grow with complex per-packet work.
  - **Queueing delay** d_{queue} is the time the packet waits in the output queue before service; it depends on the instantaneous backlog and traffic intensity on that link, so it varies widely and dominates under congestion.
  - **Transmission delay** d_{trans} is the time to push all L bits of the packet onto the link at rate R: $d_{\text{trans}} = L/R$. It depends only on packet size and link speed.
  - **Propagation delay** d_{prop} is the time for the signal to travel the physical medium from sender to next hop: $d_{\text{prop}} = d/s$, where d is the link length and s is the propagation speed in the medium (≈2×10^8 m/s in fiber).
  - Intuitively: processing is "think time," queueing is "waiting your turn," transmission is "putting bits on the wire," and propagation is "travel time over distance."

- *Explain what is the bandwidth.*
  - **Bandwidth** is the capacity of a communication channel to carry information per unit time. In packet networks we usually mean data-rate capacity, measured in bits per second (bps): a link with 1 Gbps bandwidth can transmit at most 10^9 bits each second under ideal conditions. This is distinct from throughput, which is the actual achieved rate after protocol overheads, contention, and losses; throughput $\leq$ bandwidth, and it typically varies over time as queues build and drain.
  - In signal-processing terms, bandwidth also refers to the width of the frequency band a channel occupies, measured in hertz (Hz). Wider frequency bandwidth generally allows higher potential data rates, as captured (in simplified form) by Shannon's capacity formula $C = B\log_2(1 + \text{SNR})$, where C is the theoretical maximum bit rate, B the frequency bandwidth, and SNR the signal-to-noise ratio. In practice, protocol

overhead, framing, error control, and medium access reduce usable payload rate below this limit.
- Operationally, when an offered load L approaches link bandwidth C, queueing delay grows and packets are dropped once buffers fill; this is why low-latency systems avoid persistent operation near $L \approx C$ and why adding bandwidth or smoothing traffic bursts can reduce delay even if peak throughput requirements are unchanged.

- *Can you explain the difference between bandwidth and throughput?*
  - Bandwidth is the *capacity* of a link—the maximum rate the medium can carry bits under ideal conditions—usually quoted by the provider (e.g., 1 Gbps). Throughput is the *actual* data rate you achieve end-to-end at some moment, after all real-world effects (protocol overheads, contention, congestion control, losses, CPU limits, disk I/O, etc.). Throughput is therefore always ≤ bandwidth and varies over time.
  - A handy way to think about it is: throughput ≈ (useful bytes delivered / second) = efficiency × bandwidth, where "efficiency" folds in overheads and bottlenecks along the path. For TCP specifically, the sender is also limited by how many bytes it's allowed in flight: instantaneous throughput ≈ min(cwnd, rwnd, BDP)/RTT.
  - Concrete example: you lease a 1 Gbps link (bandwidth = 10^9 b/s). With standard Ethernet framing, IP/TCP headers, and typical TCP behavior over a 20 ms RTT, your steady application throughput might be ~800–900 Mb/s under good conditions—and much less if there's loss or many small packets. The link *can* carry 1 Gb/s, but what you *get* (throughput) depends on the stack and the workload.

- *Explain what is bandwidth flooding.*
  - **Bandwidth flooding** is a denial-of-service tactic where an attacker overwhelms a victim's network links with more traffic than those links can carry, leaving too little capacity for legitimate packets. If a site's upstream capacity is C and normal traffic averages L, an attack that injects A such that $A + L > C$ forces queues to build and packets to drop; to outsiders the service looks "down" even though the servers may still be healthy.
  - Attackers typically achieve this with distributed sources (botnets) or with reflection/amplification. In a reflection attack, the attacker spoofs the victim's IP as the source of small requests to open resolvers or services (e.g., DNS, NTP, CLDAP, memcached); those services "reflect" far larger replies to the victim, creating amplification factors from single-digits up to hundreds, so modest outbound bandwidth at the attacker translates to huge inbound bandwidth at the target. Direct floods also

exist (e.g., UDP/ICMP floods or high-pps TCP floods), but modern volumetric events are often reflection-based because they scale cheaply.
- Symptoms include high packet loss, soaring latency, and link utilization pinned near 100%, often visible first at the network edge or upstream provider rather than on application hosts. Effective mitigation focuses on absorbing or diverting the volume before it reaches the bottleneck: anycasted scrubbing centers that filter by signatures and rate-limits, BGP Flowspec or ACLs at the provider to drop reflective payloads, RPKI/BCP-38 style anti-spoofing to reduce the pool of reflectors, and architectural headroom such as overprovisioned transit, caching, and separating critical control paths onto distinct links. On the host side you still harden against secondary effects—conntrack exhaustion, SYN queue pressure, and log amplification—but true bandwidth flooding is fundamentally a capacity problem that must be solved in the network, not only on the server.

- *Explain what is connection flooding.*
  - **Connection flooding** is a denial-of-service technique that targets a server's finite per-connection resources rather than the raw link capacity. Instead of saturating bandwidth, the attacker creates—or pretends to create—so many concurrent connections that the server's accept backlog, file descriptors, per-flow state (TCP control blocks, TLS handshakes, HTTP workers), or application thread pools are exhausted, causing new clients to be delayed or refused.
  - At the transport layer the classic case is a TCP SYN flood: the attacker blasts SYNs, forcing the server to allocate half-open connection state and fill the SYN backlog before the 3-way handshake completes; legitimate SYNs then time out or are dropped. A variant is a "full connection" flood that completes the handshake but immediately idles or churns connects/disconnects to burn file descriptors and per-socket memory. At the application layer, connection floods show up as many idle HTTP/TLS connections (or very slow request senders, e.g., Slowloris) that monopolize workers or keep-alive slots so real users cannot be served. Because each connection consumes kernel and user-space state, relatively modest packet rates can cause outsized damage.
  - Mitigation focuses on refusing or offloading connection state until the client proves legitimacy. For TCP, defenses include SYN cookies or SYN proxies to avoid allocating state before the handshake is validated, tightening backlog and retransmission policies, per-IP rate limits, and filtering obvious spoofed traffic upstream. For TLS/HTTP, set conservative per-IP and per-ASN connection caps, bound keep-alive lifetimes and headers/body timeouts, prefer event-driven servers over one-thread-per-connection, and terminate TLS behind load balancers or reverse proxies that can

absorb handshakes at scale. When attacks are large or distributed, upstream controls (ACLs, BGP Flowspec, scrubbing services) are needed. Conceptually, bandwidth flooding overwhelms capacity C ($A + L > C$), whereas connection flooding overwhelms state S (new or concurrent connections $> S_{\max}$); robust systems plan headroom for both and enforce backpressure before the limits are reached.

- *What is UDP?*
  UDP (User Datagram Protocol) is a lightweight, connectionless communication protocol in the Internet protocol suite that allows applications to send messages, called datagrams, without requiring a formal connection. Unlike TCP, UDP does not guarantee delivery, ordering, or error checking, making it faster and more efficient for scenarios where speed is critical and occasional data loss is acceptable. This makes UDP a popular choice for real-time applications such as live video and audio streaming, online gaming, and Voice over IP (VoIP), where minimizing latency is crucial and the overhead of establishing and maintaining a connection is undesirable.

- *What happen, let's say in a video stream, if some packet are lost in an UDP connection, how does the program recover from that?*
  UDP itself provides no mechanism for recovering lost packets—if some packets are lost in a video stream transmitted over UDP, they simply never arrive at the destination. Instead, recovery and error handling must be implemented at the application level. For example, video streaming applications often use higher-level protocols (like RTP) that add sequencing information and timestamps, allowing the application to detect missing packets and apply techniques such as error concealment, frame interpolation, or requesting keyframe updates to mitigate the impact of packet loss. Essentially, the application is responsible for designing a strategy to maintain a smooth viewing experience despite the unreliability inherent in UDP transmissions.

- *What would a loss of packet look like in a video stream?*
  In a video stream, a lost packet might manifest as visual artifacts in the displayed video. For example, you might see a temporary glitch where a portion of a frame is corrupted, a part of the image is missing, or there could be a sudden blocky or frozen area until the next complete frame arrives. Depending on how the streaming application handles error recovery (like using error concealment or interpolation techniques), the glitch might be barely noticeable or quite pronounced. Ultimately, because video streams often tolerate

occasional loss to minimize latency, these artifacts are typically designed to be transient rather than causing a prolonged disruption of the viewing experience.

- What is TCP?
  - TCP (Transmission Control Protocol) is a streaming-based protocol in the Internet protocol suite that provides a reliable, ordered, and error-checked delivery of a stream of data between applications running on hosts communicating over an IP network. Before any data is exchanged, TCP establishes a connection through a three-way handshake, ensuring that both the sender and the receiver are ready to transmit. Once connected, TCP handles data transfer by breaking the data into segments, numbering them, and ensuring they are reassembled in the correct order, while retransmitting any lost or corrupted segments. This approach helps guarantee data integrity and reliability, making TCP suitable for applications where accurate and complete data delivery is crucial, such as web browsing, email, and file transfers.
  - TCP also breaks long messages into shorter segments and provides a congestion-control mechanism, so that a source throttles its transmission rate when the network is congested.
  - Unlike UDP, TCP incorporates flow control and congestion control mechanisms to manage data transmission rates and ensure network stability, albeit at the cost of higher latency. These built-in controls prevent network congestion and help optimize resource utilization in the network, but they introduce overhead that may not be acceptable in real-time or latency-sensitive applications. The robust features of TCP—establishing reliable connections, ensuring ordered delivery, error-checking, and congestion management—make it the protocol of choice when data reliability is paramount despite the added complexity and potential delays.
  -

- *What is the difference between flow control and congestion control?*
  - **Flow control** protects the **receiver**; congestion control protects the **network**. Flow control is end-to-end backpressure that keeps a fast sender from overrunning a slow receiver's buffer. In TCP it's the **advertised receive window (rwnd)**: the receiver reports how many bytes it can still accept, and the sender limits in-flight data to that amount, independent of the path's capacity.
  - **Congestion control**, by contrast, keeps the aggregate offered load from exceeding the **path's** carrying capacity and melting down in queues and loss. In TCP it's the sender's **congestion window (cwnd)** and algorithms such as slow start, AIMD (Reno), CUBIC, BBR, and ECN: the sender probes available bandwidth and backs off

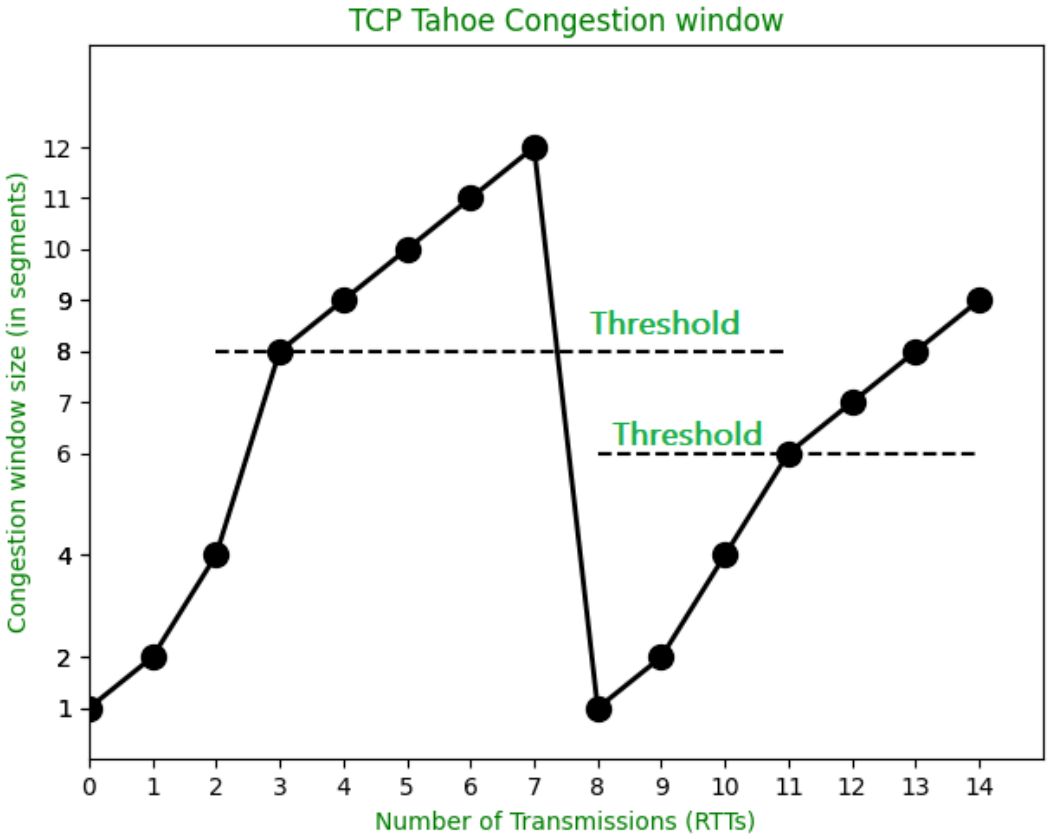
on congestion signals (loss/marks/RTT inflation), limiting in-flight data even if the receiver could take more.
    - Two quick implications follow. Flow control reacts to **receiver state** (buffer space) and can throttle a flow even on an empty network; congestion control reacts to **network state** (queueing/loss) and can throttle a flow even when the receiver has plenty of buffer. In TCP the actual sending window is the minimum of the two budgets, so the instantaneous rate is roughly $\min(\mathrm{rwnd}, \mathrm{cwnd})/\mathrm{RTT}$.

- *What kind of congestion-control mechanism is used in TCP?*
    - TCP uses **end-to-end, congestion-based control** centered on a sender-maintained **congestion window (cwnd)** and **ACK clocking**. The classic **Van Jacobson mechanism** is **slow start + Additive Increase/Multiplicative Decrease (AIMD) congestion avoidance** with **fast retransmit/fast recovery**:
        - starting from a small cwnd, the sender grows cwnd exponentially (slow start) until it hits a threshold $\mathrm{ssthresh}$,
        - then linearly increases cwnd by roughly one Maximum Segment Size (MSS) per Round-Trip Time (RTT) (additive increase).
    - If congestion occurs, TCP reduces the congestion window. The sender detects congestion when a packet needs retransmission, either due to an **RTO timeout** or **receiving three duplicate ACKs** and triggers a multiplicative decrease (cut cwnd, typically by half), plus retransmission; **fast retransmit** and **fast recovery** avoid falling back to slow start on isolated losses.
    - Modern stacks keep the same control loop but swap the increase rule: Linux defaults to **CUBIC** (a cubic growth function better suited to high-BDP paths), and alternatives like **BBR** pace to the estimated bottleneck bandwidth and RTT rather than treating loss as the primary signal. TCP can also use **ECN** to react to explicit congestion marks instead of waiting for loss.
    - See this video for more info.

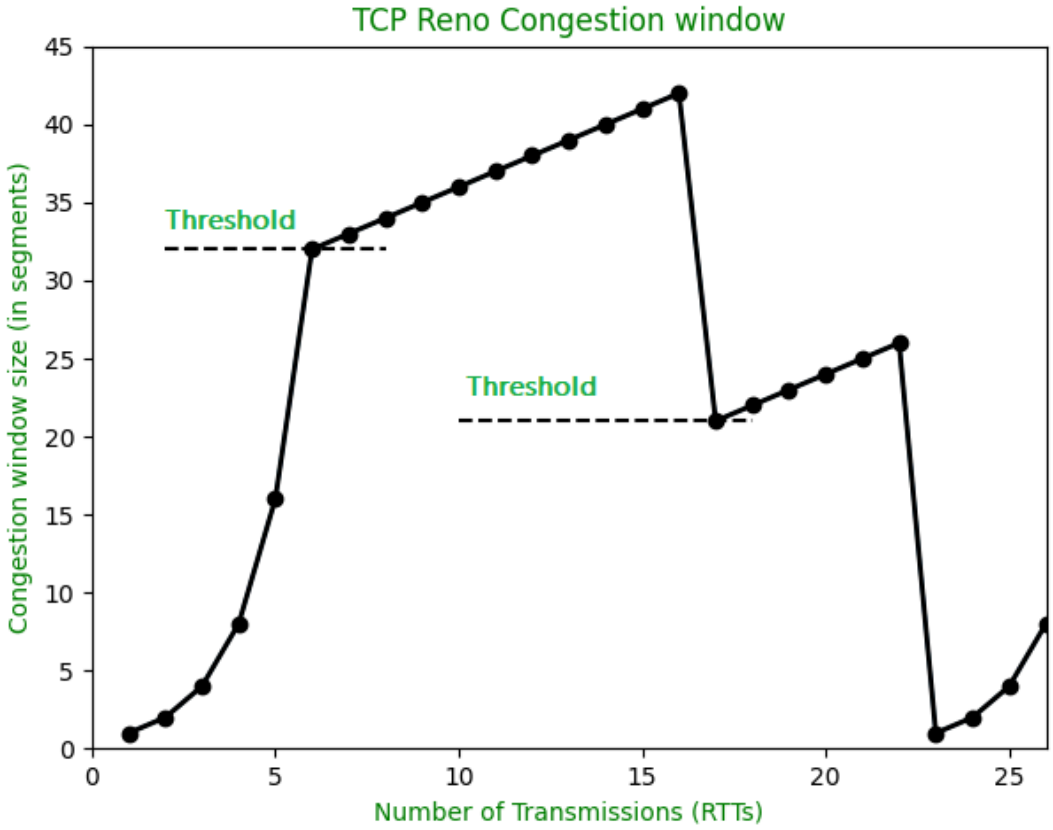
- *Explain the Van Jacobson mechanism with an example.*
    - Let MSS be the segment size, cwnd the congestion window (bytes), ssthresh the slow-start threshold, and b the ACK ratio (≈2 with delayed ACKs).
    - When an ACK for new data arrives,
        - if cwnd < ssthresh (slow start), increase cwnd by MSS for each ACK; this doubles cwnd every RTT.

    - If cwnd ≥ ssthresh (congestion avoidance), increase cwnd by roughly 1 MSS per RTT by doing a tiny increment per ACK: cwnd ← cwnd + (MSS² / cwnd).
    - If the receiver uses delayed ACKs (acks every b segments), scale the per-ACK increment to cwnd ← cwnd + (MSS² / (b·cwnd)), which still nets ≈+1 MSS per RTT.
  - When a duplicate ACK arrives (same ACK number as before), increment a dupACK counter. On the 3rd dupACK, enter fast retransmit/fast recovery: set ssthresh ← max(2·MSS, ⌊cwnd/2⌋), set cwnd ← ssthresh + 3·MSS (modeling that three segments are “leaving” the network), and immediately retransmit the presumed lost segment. For each additional dupACK beyond 3, cwnd ← cwnd + MSS and, if the congestion window allows, transmit one new segment to keep the ACK clock running. When an ACK that covers the retransmitted (previously missing) data finally arrives (a “recovery ACK”), exit fast recovery by setting cwnd ← ssthresh and clearing the dupACK counter; continue in congestion avoidance from there.
  - When the retransmission timer (RTO) fires (no progress), treat it as severe congestion: set ssthresh ← max(2·MSS, ⌊cwnd/2⌋), set cwnd ← 1·MSS, clear the dupACK counter, retransmit the oldest unacknowledged segment, and re-enter slow start (cwnd will grow to ssthresh and then switch to congestion avoidance).
  - A tiny numeric sketch with MSS=1 KB, b=2, initial cwnd=1 KB, ssthresh=16 KB: during slow start, each ACK adds 1 KB, so cwnd goes 1→2→4→8→16 KB over ~4 RTTs; now in congestion avoidance, each ACK adds ≈(1 KB)²/16 KB=64 B, so over one RTT of ~16 ACKs you gain ≈1 KB. If a loss yields 3 dupACKs at cwnd≈20 KB, set ssthresh=10 KB, cwnd=10 KB+3 KB=13 KB, fast-retransmit the missing segment, add 1 KB per extra dupACK, and when the recovery ACK arrives, drop cwnd to 10 KB and resume linear growth. If instead you hit an RTO, drop to cwnd=1 KB and slow-start back up toward ssthresh.
  - See this video from 6:40 onward and this for more info.

- *What are the different congestion-control algorithms available?*
  - TCP Tahoe and TCP Reno use the same core pieces—slow start, congestion avoidance, and fast retransmit—but they differ in what they do **after** fast retransmit.
  - **Tahoe** implements slow start + AIMD congestion avoidance + fast retransmit **without** fast recovery. On 3 duplicate ACKs, Tahoe assumes a loss, sets `ssthresh = ⌊cwnd/2⌋`, **drops `cwnd` to 1 MSS**, retransmits the missing segment, and re-enters **slow start** until `cwnd` reaches `ssthresh`, after which it continues with linear (AIMD) growth. A timeout does the same (set `ssthresh = ⌊cwnd/2⌋`, `cwnd = 1 MSS`).



- **Reno** adds **fast recovery** to avoid the big collapse on isolated loss. On 3 duplicate ACKs, Reno sets `ssthresh = ⌊cwnd/2⌋`, sets **`cwnd = ssthresh + 3·MSS`** (modeling the three dupACKed segments still in flight), **retransmits immediately**, and stays out of slow start. For each additional dupACK, it increments `cwnd` by 1 MSS (sending a new segment if allowed) to keep the ACK clock running. When the ACK that covers the retransmitted segment arrives, Reno **exits fast recovery** by setting `cwnd = ssthresh` and continues in congestion avoidance. A timeout still forces `cwnd = 1 MSS` (like Tahoe).



- **CUBIC** replaces Reno's linear AIMD with a cubic growth law designed for high-BDP links and reduced RTT bias. After a loss, let $W_{\max}$ be the window just before loss. CUBIC sets a target window

$$\mathrm{cwnd}(t) = C\,(t-K)^3 + W_{\max}, \quad K = \sqrt[3]{\frac{W_{\max}\,(1-\beta)}{C}},$$

  where t is time since the last loss, C is a constant (≈0.4), and β is the multiplicative decrease factor (≈0.7). The curve grows slowly near $W_{\max}$ (to probe carefully), then faster as it moves away, making the increase depend mainly on elapsed time rather than RTT. On loss, CUBIC reduces cwnd by $\beta \cdot \mathrm{cwnd}$ (similar spirit to Reno's halving but with $\beta \neq 0.5$). In practice CUBIC achieves higher utilization on fast long-distance paths and is Linux's long-time default; its trade-offs include weaker RTT fairness versus Reno and sensitivity to very shallow buffers.
- **BBR/BBRv2** departs from loss-based AIMD entirely: it models the path and paces at the estimated bottleneck bandwidth times the (minimum) RTT, i.e., targets an in-flight budget $\mathrm{BDP} \approx \mathrm{BtlBw} \times \mathrm{RTT}_{\min}$. The sender sets a **pacing rate** $\approx \mathrm{BtlBw} \times \mathrm{gain}$ and a **cwnd cap** $\approx \mathrm{BDP} \times \mathrm{gain}$, cycling through phases: **Startup** (aggressive gain to discover capacity), **Drain** (shed any queue formed), and a steady-state **ProbeBW** that varies the gain to gently explore for more bandwidth, plus periodic **ProbeRTT** to refresh $\mathrm{RTT}_{\min}$. Classic BBR (v1) can be too optimistic in some mixes; **BBRv2** adds explicit loss/ECN response, tighter inflight ceilings, and fairness improvements so it

coexists better with CUBIC/Reno and behaves well with shallow buffers. The result is good throughput at low queueing delay on clean or long-fat paths; the trade-off is greater complexity and the need for careful pacing/timer accuracy.

- Practical implications: Reno performs better than Tahoe when there's **a single loss per window** (it avoids the full slow-start restart), but classic Reno struggles with **multiple losses in one RTT** (often falling back to timeout). **CUBIC** is time-based cubic probing (great on high-BDP paths; widely deployed). **BBR/BBRv2** is model-based pacing to $\mathrm{BtlBw} \times \mathrm{RTT}_{\min}$ (aims for high throughput with low latency by avoiding standing queues). Each still uses TCP's slow start at the beginning; they diverge in how they grow, probe, and cut back once past slow start.
- See this for more info.

- *Which algo is typically used by HFT firms?*
  - There isn't a single "HFT congestion-control algorithm." In practice:
    - **Order entry/control (TCP in the colo):** links are short, clean, and run far below capacity. Congestion control almost never triggers, so firms usually leave the **OS default (Linux CUBIC)** or sometimes **NewReno** for predictability. What matters much more is `TCP_NODELAY`, tiny socket buffers, pinning, and eliminating coalescing—*not* the CC math.
    - **Market data:** it's **UDP multicast** (no TCP CC at all).
    - **Long-haul/backups/bulk** paths: some shops may try **BBR/BBRv2** for high-BDP links, but that's outside the latency-critical loop.
  - Rule of thumb: in the hot path, **CUBIC (default) is fine and common; BBR is rare** for HFT order traffic because its pacing/measurement focus targets throughput and queue control, not microsecond request/response. The performance wins in HFT come from stack/bypass choices, NIC tuning, and application design—not from switching Reno↔CUBIC↔BBR.

- *What is the difference between order entry and market data?*
  - **Order entry** is the traffic you send **to** the exchange to act (new orders, cancels, modifies, risk/admin); market data is the firehose you receive **from** the exchange describing the market state (quotes, trades, order-book updates). Order entry is almost always **TCP**, sessionful, and request/response with strict sequencing per session (e.g., FIX, proprietary binary gateways). It prioritizes determinism and immediate acknowledgments, uses `TCP_NODELAY`, and is governed by exchange **rate**

limits/throttles and risk checks—if you overrun limits, the exchange may pause or reject messages. Bandwidth is modest but latency and predictability are paramount.

- **Market data** is typically **UDP multicast** (e.g., ITCH/MDP): the exchange publishes one stream to many subscribers; there's **no built-in reliability or backpressure**. Messages carry **sequence numbers**; your feed handler must detect gaps, request **retransmissions** via a separate service or take a snapshot, and deliver updates to your strategies strictly in order. Feeds are high-pps, bursty (especially at opens), engineered for fan-out and minimal publisher latency, often with **A/B redundant networks** for resilience. In short: order entry = low-latency, stateful **commands over TCP** with exchange-enforced pacing; market data = high-rate, best-effort **telemetry over UDP multicast** with client-side recovery.

- *What kind of congestion control is used by exchanges like CBOE?*
  - Short answer: for market-data, **none**; for order-entry, **ordinary TCP congestion control** (e.g., CUBIC) plus **exchange-side rate limits/throttles**.
  - More detail: Cboe's market-data feeds (e.g., Multicast PITCH) are sent over **UDP multicast**, which deliberately has **no end-to-end congestion control**. Reliability is handled out-of-band: messages carry sequence numbers; if you detect a gap you recover via Cboe's **Gap Request Proxy (GRP)** over TCP, or by taking a TCP "spin"/snapshot service; venues also run redundant A/B multicast networks. The feed itself isn't slowed by receiver congestion. CBOE+2CBOE+2
  - For **order entry and control**, firms connect to Cboe gateways over **TCP**, so congestion control is whatever the host OS implements (on Linux, typically **CUBIC**, unless a firm switches to BBR, etc.). Separately—and more importantly in practice—Cboe enforces **message-rate limits/throttles** at the port/session level (e.g., per-port order rate thresholds, pause-on-violation, limits for mass-cancel/purge), which act like admission control rather than transport-layer congestion control. CBOE+3res-certification.cboe.com+3Cboe Global Markets+3
  - So the right mental model is: exchanges engineer feeds to be fast and **statistically multiplexed** without sender back-off, and they provide deterministic **recovery services**; for order traffic they rely on **TCP's CC** plus **explicit exchange throttles** to keep gateways healthy.

- *What is the TCP sending rate and how do you calculate it?*
  - In TCP, the instantaneous **sending rate** (application goodput) is essentially the amount of data the sender is allowed to have "in flight" divided by the round-trip time.

In symbols, with congestion window cwnd (bytes) and round-trip time RTT,

$$\text{rate} \;\approx\; \frac{\text{cwnd}}{\text{RTT}} \quad \text{(bytes/s)}.$$

Because TCP is ACK-clocked, cwnd grows/shrinks with the congestion-control algorithm, and the actual rate is bounded by the minimum of three limits:

$$\text{rate} \;\approx\; \frac{\min(\text{cwnd},\, \text{rwnd},\, \text{BDP})}{\text{RTT}},$$

where rwnd is the receiver window and $\mathrm{BDP} = C \cdot \mathrm{RTT}$ (the path's bandwidth–delay product) caps what the path can carry. Using packet units with MSS (bytes per segment), you'll often see

$$\text{rate} \;\approx\; \frac{\text{cwnd} \,/\, \text{MSS}}{\text{RTT}} \cdot \text{MSS}.$$

For long-lived flows under Reno-like control with random loss probability p, a widely used steady-state model (Mathis et al.) gives the average throughput:

$$\boxed{\text{rate} \;\approx\; \frac{1.22\,\text{MSS}}{\text{RTT}\,\sqrt{p}}}$$

(roughly, AIMD grows linearly until a loss halves cwnd, producing the $1/\sqrt{p}$ dependence). Modern algorithms differ: CUBIC uses a cubic growth law (throughput depends weakly on RTT), and BBR targets $\text{rate} \approx \text{bottleneck bandwidth}$ bandwidth by pacing to measured bandwidth and RTT rather than loss, but the instantaneous TCP sending rate is still governed by "allowed bytes in flight over RTT."

- *What are the mechanisms in place in TCP to provide error-free transmission of data?*
  - TCP employs several mechanisms to ensure error-free data transmission:
    1. **Checksum:**
    Every TCP segment includes a checksum that covers both the header and the data payload. The receiver uses this checksum to detect any corruption that might have occurred during transmission. If the checksum doesn't match, the segment is considered corrupted. In detail, the checksum is derived directly from the original message using some well-defined algorithm. Both the sender and the receiver know exactly what that algorithm is. The sender applies the algorithm to the message to generate the redundant bits. It then transmits both the message and those few extra bits. When the receiver applies the same algorithm to the received message, it should (in the absence of errors) come up with the same result as the sender. It compares the result with the one sent to it by the sender. If they match, it can conclude (with high

likelihood) that no errors were introduced in the message during transmission. If they do not match, it can be sure that either the message or the redundant bits were corrupted, and it must take appropriate action—that is, discarding the message or correcting it if that is possible. One note on the terminology for these extra bits follows. In general, they are referred to as **error-detecting codes**. In specific cases, when the algorithm to create the code is based on addition, they may be called a **checksum**.

2. **Sequence Numbers and Acknowledgments:**
Each byte in the TCP data stream is assigned a sequence number. The sender and receiver use these numbers to keep track of which data has been sent and received. The receiver sends back acknowledgments (ACKs) indicating the next expected sequence number. If the sender doesn't receive an acknowledgment within a certain time, it assumes that the segment was lost or corrupted and retransmits it.

3. **Retransmission:**
TCP automatically retransmits segments if acknowledgments are not received within a specified timeout period. This retransmission mechanism ensures that lost or corrupted segments are resent until they are correctly received.

4. **Flow Control (via the Window Size):**
The receiver advertises a window size that tells the sender how many bytes it is prepared to accept at a time. This helps prevent buffer overruns and ensures that the sender doesn't overwhelm the receiver, which could lead to errors.

5. **Congestion Control:**
TCP monitors network congestion through various algorithms (such as slow start, congestion avoidance, fast retransmit, and fast recovery). These mechanisms adjust the rate at which data is sent to match current network conditions, reducing the likelihood of packet loss due to congestion.
These combined mechanisms allow TCP to provide a reliable, ordered, and error-checked data delivery service even over networks where packet loss, corruption, and delay may occur.

- *What is the difference between flow control and congestion control?*
  - Flow control protects the **receiver**; congestion control protects the **network**.
  - Flow control is end-to-end backpressure from the **receiver's buffer capacity**. In TCP it's the **advertised receive window (rwnd)**: the sender limits in-flight data so it doesn't overrun the receiver, regardless of path capacity.

- Congestion control limits the sender based on **path conditions** (queueing, loss, ECN marks, RTT inflation) to avoid overloading the network. In TCP it's the **congestion window (cwnd)** and algorithms like slow start, AIMD/Reno, CUBIC, BBR.
- At any instant a TCP sender's rate is bounded by both:

  $$\text{throughput} \approx \frac{\min(\text{cwnd}, \text{rwnd})}{\text{RTT}}.$$

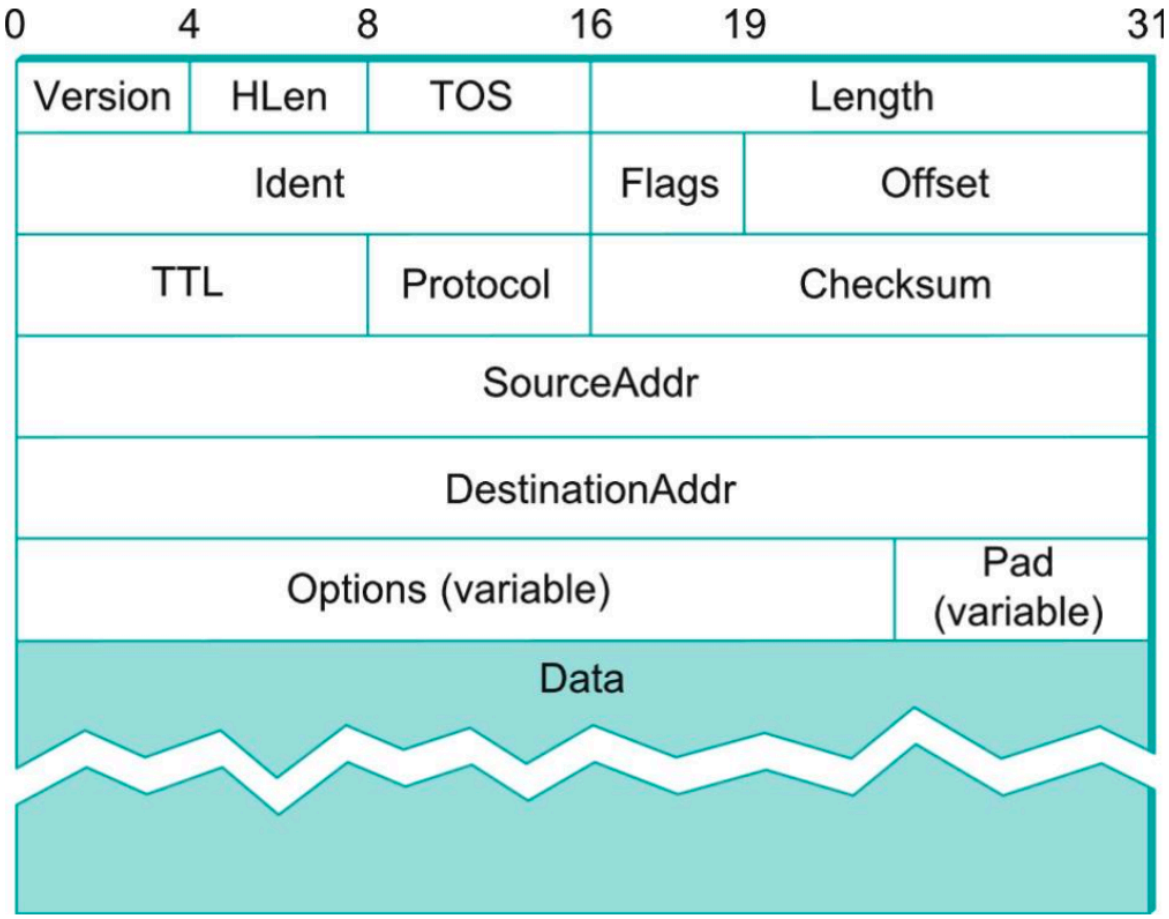
  So flow control reacts to receiver state; congestion control reacts to network state.

- *Can you elaborate on what is a TCP segment ? Is it a single packet?*
  - A **TCP segment** is the basic unit of data that TCP prepares for transmission, and it consists of a header and a payload. The header contains crucial information such as source and destination ports, sequence and acknowledgement numbers, flags (like SYN, ACK, FIN), a checksum, and other fields used for managing the connection and ensuring reliable delivery. Although it's often casually referred to as a "packet," a TCP segment specifically pertains to the transport layer; when it is sent over the network, it becomes encapsulated within an IP packet. So while every TCP segment eventually sits inside an IP packet for transmission over the network, the term "TCP segment" focuses on the structure and content defined by the TCP protocol itself.

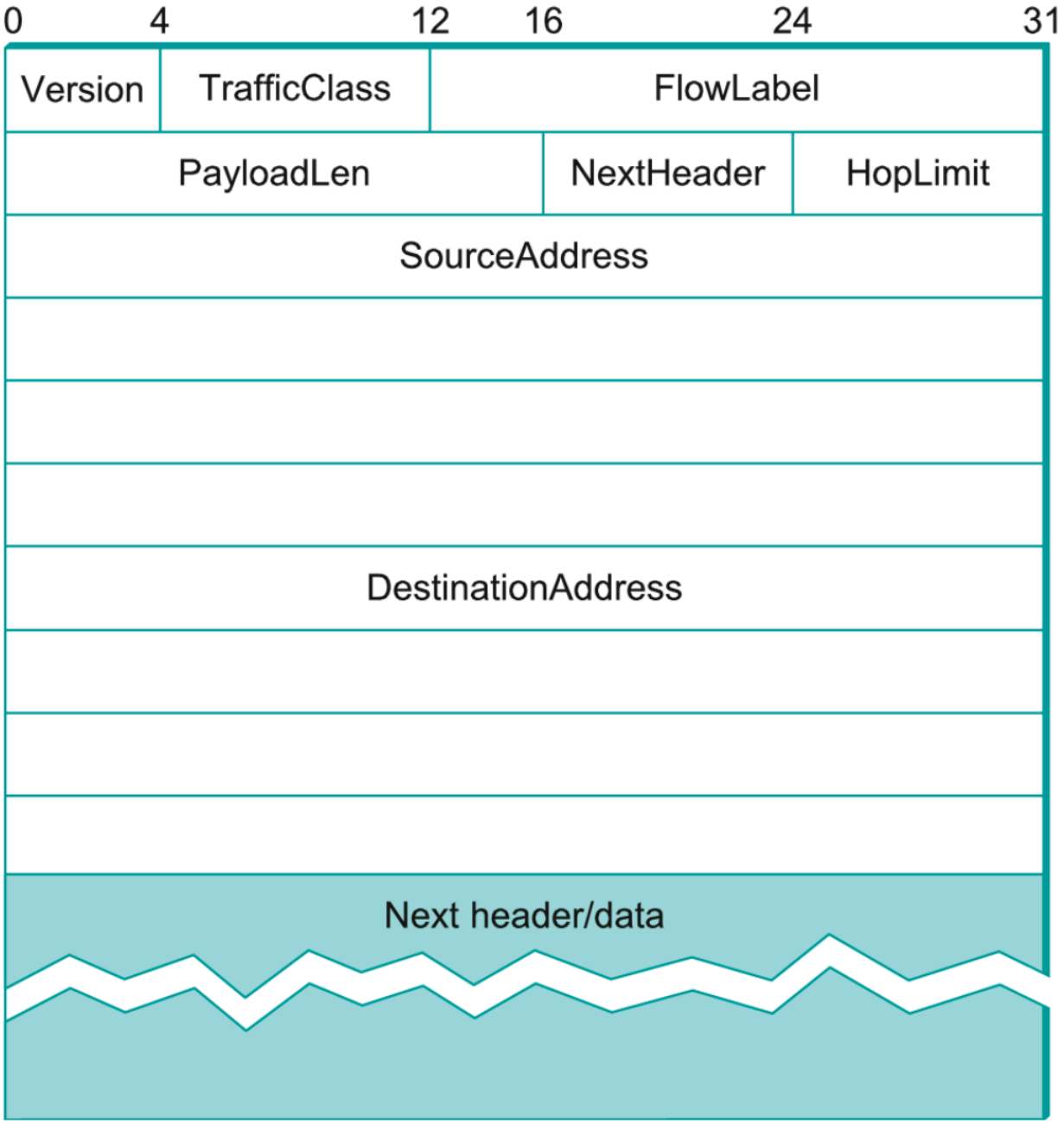
- *What are the size of TCP and UDP headers?*
  - In networking:
    - **UDP (User Datagram Protocol)** has a **fixed header size of 8 bytes**.
      - It contains only four fields, each 2 bytes long:
        1. Source Port
        2. Destination Port
        3. Length (Header + Payload in bytes)
        4. Checksum
      - So the **UDP header = 8 bytes total**.
    - **TCP (Transmission Control Protocol)**, on the other hand, has a **variable header size**. The **minimum** TCP header size is **20 bytes**, but it can grow up to **60 bytes** if options (like timestamps, window scaling, etc.) are included.

- *What are the sizes of IPv4 and IPv6 headers?*

- **IPv4 header: 20 bytes** minimum (no options), **up to 60 bytes** with options.



- **IPv6 header: 40 bytes** fixed (extensions are separate headers that add beyond 40).



- *What is a three-way handshake?*
  - The **three-way handshake** is the process used by TCP to establish a reliable connection between a client and a server before any actual data transfer occurs. It involves three steps: First, the client sends a SYN (synchronize) segment to the server to initiate the connection and inform the server of the client's initial sequence number. Next, the server responds with a SYN-ACK (synchronize-acknowledge) segment, which acknowledges the client's SYN and includes its own SYN request with the server's initial sequence number. Finally, the client sends an ACK (acknowledge) segment back to the server, confirming the receipt of the server's SYN-ACK. Once this

three-step exchange is complete, both sides have synchronized sequence numbers and the connection is fully established, enabling reliable data transfer over the network.

- What are SYN and ACK?
  - **SYN (synchronize)** and **ACK (acknowledge)** are control flags in the TCP header that play critical roles in establishing and managing a TCP connection.
  - The **SYN** flag is used when initiating a connection; it signals the intent to establish communication and is often accompanied by an initial sequence number, which helps in synchronizing the communication between the sender and receiver.
  - The **ACK** flag is used to confirm the receipt of data or control information—in the context of a TCP handshake, it acknowledges receipt of a SYN or other data segments.
  - Together, these flags are used in the three-way handshake process: a client sends a SYN to start the connection, the server replies with a SYN-ACK to both acknowledge the client's request and send its own synchronization request, and finally, the client sends an ACK to confirm the server's response, thereby completing the connection setup.

- *Can you explain what is the sliding window protocol?*
  - The sliding-window protocol lets a sender transmit **multiple, sequentially numbered frames** without waiting for an ACK after each one, while ensuring reliable, in-order delivery and **flow control**. The sender maintains a *window* over a range of sequence numbers: all frames inside the window may be “in flight.” As ACKs arrive (usually cumulative), the window **slides forward**, permitting new transmissions. The receiver keeps its own window of frames it is willing to accept and buffer, advertising this as a **receive window** to keep the sender from overrunning its buffers.
  - TCP is a sliding-window protocol layered over IP. Its *effective* in-flight data is limited by the **minimum of** the receiver’s advertised window (**rwnd**, flow control) and the sender’s **congestion window** (**cwnd**, congestion control):

    $$\text{in flight} \leq \min(\text{rwnd},\ \text{cwnd}).$$

    TCP uses **cumulative ACKs** (optionally SACK for SR-like efficiency), **ACK clocking** to pace sends, retransmission timers and fast retransmit/fast recovery for loss, and window updates from the receiver.

- Rule of thumb for throughput on a stable path: with window W (bytes) and RTT R, the sending rate is roughly W/R. Larger windows increase pipeline depth and utilization; too large without control creates queues and loss. Sliding windows strike the balance by keeping the pipe full while bounding the number of outstanding, unacknowledged frames.

- Can you show what a TCP segment contains and would look like?
  - A **TCP segment** is composed of a TCP header and, optionally, payload data. The header contains all the necessary control information to manage the transmission and ensure reliability. Here's an example of what a TCP segment header might include:
    - **Source Port (16 bits):** The port number of the sender.
    - **Destination Port (16 bits):** The port number of the receiver.
    - **Sequence Number (32 bits):** Used to identify the order of data bytes sent.
    - **Acknowledgment Number (32 bits):** Indicates the next byte expected by the sender of the segment.
    - **Data Offset (4 bits):** Specifies the size of the TCP header.
    - **Reserved (3 bits):** Reserved for future use and typically set to zero.
    - **Flags (9 bits):** Control bits, including:
      - **NS (1 bit):** ECN-nonce concealment protection.
      - **CWR (1 bit):** Congestion Window Reduced.
      - **ECE (1 bit):** ECN-Echo.
      - **URG (1 bit):** Indicates urgent data.
      - **ACK (1 bit):** Indicates that the acknowledgment field is valid.
      - **PSH (1 bit):** Push function.
      - **RST (1 bit):** Reset the connection.
      - **SYN (1 bit):** Synchronize sequence numbers (used to initiate a connection).
      - **FIN (1 bit):** No more data from the sender.
    - **Window Size (16 bits):** The size of the sender's receive window (i.e., the amount of data that can be accepted).
    - **Checksum (16 bits):** Used for error-checking of the header and data.
    - **Urgent Pointer (16 bits):** Points to the end of urgent data if the URG flag is set.
    - **Options (variable length):** Optional settings (e.g., maximum segment size, window scaling) padded so that the header length is a multiple of 32 bits.
    - **Data (payload):** The actual data being transmitted.
  - A simplified ASCII diagram of a TCP segment header might look like this:

```
0 1 2 3 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----------------------------------------------------------------+ |
Source Port | Destination Port | +-----------------------------------
------------------------+ | Sequence Number | +----------------------
-------------------------------------+ | Acknowledgment Number | +----
-----------------------------------------------------------+ | Data
|Reser-|Flags| Window Size | | Offset |ved | | | +--------------------
-------------------------------------+ | Checksum | Urgent Pointer |
+-----------------------------------------------------------------+ |
Options (if any) ... | +-----------------------------------------------
--------------+ | Data (payload) ... | +-----------------------------
--------------------------------+
```

- This header is then encapsulated within an IP packet when transmitted over the network. While this diagram highlights the TCP header fields, the actual values and sizes might vary slightly depending on specific implementations and the presence of options.

- *If a network link has a capacity of 10'000 bauds (bits per seconds), how much time will serializing 1500 bytes take?*
  - First, convert the bits to bytes: $\frac{10000}{8} = 1250 \text{ bytes}$.
  - Hence, it takes 1 second to serialize 1250 bytes.
  - As a result, it takes $1 + \frac{250}{1250} = 1.20 \text{ seconds}$ to serialize 1500 bytes

---

- *Can you explain everything that happens from when a TCP packet arrives at the NIC, to the point where it reaches the user process?*
  - Here's the typical **Linux TCP receive fast-path** from wire to your `recv()` call (no kernel bypass):
    1. **NIC → DMA into RAM.** The Ethernet frame hits the NIC. The NIC verifies checksums (offload), timestamps if enabled, and **DMAs** the bytes into a pre-posted RX descriptor (a buffer in system RAM). With **RSS**, a hash of the 5-tuple chooses an RX queue (and thus a CPU/IRQ). The NIC marks the descriptor "done."
    2. **Interrupt & NAPI scheduling.** The NIC raises an **MSI-X interrupt** to the target CPU. The hard-IRQ handler does almost nothing except disable further interrupts for that queue and schedule **NAPI** (softirq). Under load, work may run in `ksoftirqd` (adds jitter); under light load it runs immediately in softirq context.
    3. **Driver poll & skb creation.** In `net_rx_action`, the driver **polls** the RX ring (up to a budget), harvests completed descriptors, and wraps each packet in an `sk_buff` (**skb**) that points at the DMA'd data (often as page fragments). Optional **GRO** may

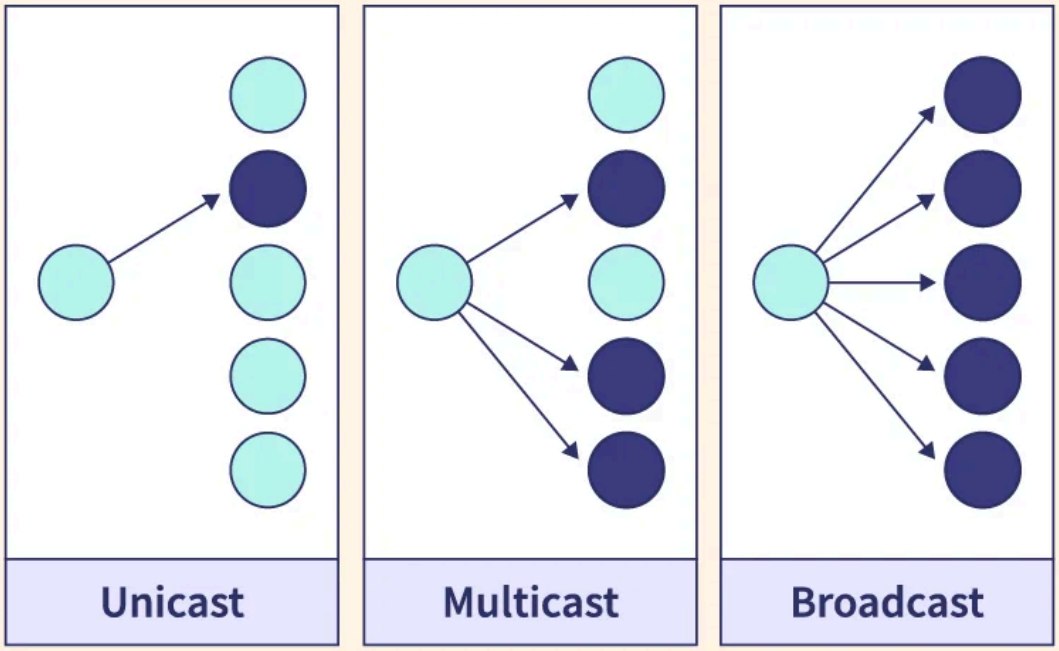
coalesce multiple same-flow TCP segments into a larger pseudo-packet to reduce per-packet overhead (great for throughput, adds a bit of latency).

4. **L2/L3 handing & hooks.** The skb goes through L2 (VLAN, EtherType) and **IP**: header parse, checksum if not offloaded, routing decision. **Netfilter**/nftables/conntrack hooks can inspect/modify/drop. Each of these touches caches and can add a little variability.
5. **TCP demux & processing.** The TCP layer looks up the **socket** by 4-tuple (plus namespace), validates sequence numbers and window, and updates the per-connection **TCB** (receive next, out-of-order queue, SACK scoreboard, timestamps). If segments are out of order, they're queued for **reassembly**; if in order, bytes advance `rcv_nxt`. TCP may generate **ACKs** (possibly **delayed ACK**) and window updates, and it honors flow control (`rwnd`) and autotuning of `SO_RCVBUF`.
6. **Enqueue to the socket receive queue.** Once TCP has **in-order bytes**, it enqueues them on the socket's **receive queue** (still in kernel memory). If your thread is sleeping in `poll/epoll` or `recv*`, the kernel marks the socket readable and **wakes** your thread. If not, data waits in the queue up to buffer limits.
7. **Scheduler wakeup & context switch.** Your thread is placed on the run queue for that CPU and eventually scheduled. If IRQ/NAPI ran on a different core than your app, you can pay cross-core/NUMA penalties; good setups **pin** the RX queue and app thread to the same core/NUMA node.
8. **Your `recv*` syscall & the copy.** In user space you call `recv()/recvmsg()/recvmmsg()`. The kernel dequeues bytes from the socket and **copies** payload from kernel memory into your user buffer (the unavoidable kernel→user copy on the normal stack). It fills address/ancillary data as needed and returns the number of bytes. With **GRO**, you may get a large read that came from many original segments.
9. **Acks, timers, bookkeeping continue.** Independently, TCP's timers (RTO, keepalive) and **ACK** generation proceed; if SACK is enabled and there were gaps, selective ACKs guide the sender's retransmissions. Congestion control (CUBIC/BBR/etc.) affects the **sender** side; on receive you mainly influence advertised window and ACK behavior.

- See this for more info.

- *Explain the difference between unicast, broadcast and multicast.*
  - Unicast, broadcast, and multicast are three different ways of delivering packets, and they mainly differ in **who** is supposed to receive the packet and **how the network forwards it**.

- Unicast is one-to-one communication. A packet has a single specific destination address, and the network's job is to deliver it to exactly that one host. Almost everything you normally do on the internet is unicast: connecting to a web server, SSH, sending an email, etc. At the IP layer this is a normal unicast address (for IPv4 something like 192.168.1.42; for IPv6 a unicast address as well), and at the Ethernet layer the frame is sent to a single unicast MAC address. Switches and routers forward it along a single path toward that host.
- Broadcast is one-to-all on a local network segment. A broadcast packet is meant for "everyone on this link," not one specific host. In IPv4 you have special broadcast addresses like 255.255.255.255 (limited broadcast) or the all-ones address of a subnet, and on Ethernet the frame goes to the broadcast MAC address ff:ff:ff:ff:ff:ff. Every device on that LAN receives and processes the packet (at least enough to decide if it cares). Protocols such as ARP and classic DHCP use broadcast so that a host can talk to "whoever is responsible" without knowing their address in advance. Broadcast does not scale across the wider internet; routers normally do not forward broadcast traffic, so its scope is basically one layer-2 domain.
- Multicast is one-to-many, but only to interested receivers. Instead of sending a separate unicast copy to each receiver, the sender addresses a multicast group. Hosts that want the data "join" that group, and the network delivers the packet only to them. In IPv4, multicast addresses live in the 224.0.0.0–239.255.255.255 range; in IPv6, multicast is built in directly and replaces broadcast entirely (there is no IPv6 broadcast). At the Ethernet layer, multicast IP addresses are mapped to special multicast MAC addresses. Routers that support multicast can build a distribution tree so that each link only carries one copy of the packet, which is then replicated where paths diverge. This makes multicast efficient for scenarios like live video streaming, stock quote distribution, or any "same data to many receivers" problem, without flooding everyone like a broadcast.



- *Can you explain what is multicast and how it works?*
  - **Multicast** is a one-to-many (or few-to-many) delivery mode where a sender transmits **one copy** of a packet and the network **replicates it only where needed** so that multiple receivers that “joined” a group all get the same data. It sits between unicast (one-to-one) and broadcast (to everyone on a link).
  - **Addressing & groups.** Receivers join a **multicast group address** (IPv4 224.0.0.0/4, IPv6 ff00::/8). A host expresses interest via **IGMP** (IPv4) or **MLD** (IPv6). Applications usually use **UDP** to send to the group IP+port.
  - **On a LAN.** Switches can either flood multicast like broadcast or (better) use **IGMP snooping** to learn which ports have members of a group and forward only there. NICs map IP multicast to a special **Ethernet MAC** (01:00:5e:… for IPv4; 33:33:… for IPv6).
  - **Across subnets.** Routers build multicast trees so the sender’s single stream is **replicated at branch points**. Common control planes are **PIM** (Protocol Independent Multicast) variants that coordinate where to forward. Two models exist:

    - **ASM (Any-Source Multicast):** receivers join a group (*G*); routers discover sources dynamically (may use Rendezvous Points).
    - **SSM (Source-Specific Multicast):** receivers join (*S,G*), i.e., a specific source *S* and group *G*—simpler, more secure, preferred where possible.
- **Joining & leaving.** A receiver calls `setsockopt(IP_ADD_MEMBERSHIP/IPV6_JOIN_GROUP)` and the host sends IGMP/MLD **joins** toward the router. Leave is the reverse. As long as at least one receiver downstream exists, the router keeps the branch and replicates packets.
- **Reliability & ordering.** IP multicast provides **no reliability, ordering, or congestion control**. If a receiver misses packets, it's up to the application to recover (e.g., sequence numbers, repairs via unicast "gap fill," or FEC). This is why exchanges use **UDP multicast** for market data and pair it with **retransmission/snapshot** services.
- **Scope & TTL.** Multicast can be scoped: some group ranges are link-local only; applications also control reach with **TTL/Hop Limit** and administrative boundaries (e.g., not routed over the public Internet by default).
- **Why use it.** The sender's **one transmit → many receivers** saves bandwidth and CPU versus sending N unicasts, and the network does replication close to the receivers. Typical uses: live video/audio, service discovery, telemetry, market-data distribution, cluster membership.

- *How does multicast saves bandwidth over N unicasts?*
    - With **unicast**, the sender emits **one separate copy per receiver**; with **multicast**, the sender emits **one copy total**, and the **network replicates it only at branch points** where paths diverge.
    - Concrete picture: you stream **10 Mb/s** to **100 receivers**.
        - **Unicast:** your uplink carries **100 × 10 Mb/s = 1,000 Mb/s**. Every shared core link also carries many duplicate copies until paths split.
        - **Multicast:** you transmit **one 10 Mb/s flow**. The first few shared links (your uplink, the core) still carry **10 Mb/s total**. Only where the tree branches does a router/switch replicate the packet for each downstream limb. If 60 receivers sit behind one aggregation router and 40 behind another, the core still sees **10 Mb/s**, each aggregation link sees **10 Mb/s**, and only the **last hop** to each access switch/host has its own copy.
    - So multicast saves bandwidth on all **common segments** of the path (especially your sender uplink and the backbone). Replication happens **close to the receivers**, not at the source. On a single L2 LAN with IGMP snooping, the switch also forwards to **only the ports that joined**, not every port. Caveats: if snooping is off, switches may flood

(no savings); and the public Internet generally doesn't route multicast, so this benefit is realized in managed networks (exchanges, ISPs, enterprises).

C++/Linux-focused Networking Questions

- *Explain what is a socket in Linux.*
  - A **socket** in Linux is a kernel-managed **endpoint for network or inter-process communication**. It's exposed to user space as a **file descriptor**, so you use the same read/write/ `poll` -style APIs you'd use for files, but the kernel routes data to/from a network (IP) or a local IPC transport (Unix domain).
  - Conceptually, a socket has three attributes: an **address family** (e.g., `AF_INET` / `AF_INET6` for IPv4/IPv6, `AF_UNIX` for local IPC), a **type** (e.g., `SOCK_STREAM` = reliable byte stream, typically TCP; `SOCK_DGRAM` = message-oriented, typically UDP), and a **protocol** (often inferred from the type). Under the hood the kernel maintains per-socket state and queues (send/receive buffers) and interacts with the NIC/stack to move bytes.
  - Lifecycle differs by role:
    - **Client (active open, e.g., TCP):** `socket()` → optional `setsockopt()` (e.g., `TCP_NODELAY`) → `connect()` ; then `send` / `recv` (or non-blocking + `epoll`) → `shutdown` / `close` .
    - **Server (passive open, e.g., TCP):** `socket()` → `setsockopt(SO_REUSEADDR{,PORT})` → `bind()` (assign local IP/port or Unix path) → `listen()` → loop `accept()` to get a new connected socket per client → I/O → `close` .
    - **Datagram (UDP):** `socket()` → optional `bind()` (to receive on a specific port/interface) → `sendto` / `recvfrom` (each call handles one datagram). Multicast adds group membership via `setsockopt(IP_ADD_MEMBERSHIP)` .
  - Behavioral semantics come from the **type**: streams are ordered, reliable byte pipes with no message boundaries; datagrams preserve message boundaries but can be lost/reordered/duplicated. Blocking vs non-blocking is a per-descriptor mode (`fcntl`), and readiness is observed with `poll/epoll` . Many details are tunable via `setsockopt` (buffer sizes, timeouts, TTL, ECN, keepalives, etc.).
  - Finally, not all sockets are "network": **Unix domain sockets** (`AF_UNIX`) provide fast, credential-aware IPC on the same machine using filesystem-namespaced endpoints; they support both stream and datagram semantics and avoid IP entirely.
  - In short, a Linux socket is a **file-descriptor interface to transport endpoints** that the kernel's networking/IPC stacks drive, giving your program a uniform way to establish connections, exchange data, and manage communication policies.

- *What is the endianess of network-byte-order? How would you deal with that on a little-endian machine in a C++ program?*
  - Network byte order is **big-endian**: multi-byte integers are transmitted most-significant byte first. On a little-endian host (e.g., x86-64), you must convert between “network” and “host” order at your program’s edges. On POSIX you do this with the standard conversion functions in `<arpa/inet.h>` — `htons/htonl` when **sending** (host→network) and `ntohs/ntohl` when **receiving** (network→host). They operate on 16- and 32-bit integers; on big-endian machines they’re no-ops, and on little-endian they perform a byte swap so your in-memory values are correct.
  - When parsing bytes from the wire, don’t alias unaligned buffers; copy into a fixed-width integer and convert. For example, reading a 16-bit “type” followed by a 32-bit “length” from a buffer:

```
#include <cstdint>
#include <cstring>
#include <arpa/inet.h>
// Windows:
<winsock2.h>  struct Header
{
    std::uint16_t type;
    std::uint32_t len;
};
void parse_header(const unsigned char* buf, Header& out)
{
    std::uint16_t type_be;
    std::uint32_t len_be;
    std::memcpy(&type_be, buf,        sizeof type_be);
    std::memcpy(&len_be,  buf + 2,    sizeof len_be);
    out.type = ntohs(type_be);
    out.len  = ntohl(len_be);
}
```

  - When emitting bytes to the wire, convert first, then copy out:

```
void write_header(unsigned char* out, std::uint16_t type,
std::uint32_t len)
{
    std::uint16_t type_be = htons(type);
    std::uint32_t len_be  = htonl(len);
    std::memcpy(out,        &type_be, sizeof type_be);
    std::memcpy(out + 2,    &len_be,  sizeof len_be);
}
```

- For 64-bit fields there isn't a portable ISO C function. On Linux/glibc and the BSDs you can use `htobe64/be64toh` (`<endian.h>` or `<sys/endian.h>`). In portable C++ you can write a tiny helper; with C++23 you get `std::byteswap` and can key off `std::endian`:

```
#include <bit>
#include <cstdint>
constexpr std::uint64_t hton64(std::uint64_t x) {
    if constexpr (std::endian::native == std::endian::little)
        return std::byteswap(x);
    else return x;
}
constexpr std::uint64_t ntoh64(std::uint64_t x) { return hton64(x); }
```

- Two pragmatic rules keep you out of trouble. First, always use fixed-width types (`std::uint16_t`, `std::uint32_t`, `std::uint64_t`) for on-the-wire fields and convert at the boundary; never `send()` a C++ struct directly (padding, alignment, and endianness make it non-portable). Second, treat network data as bytes until you explicitly convert it; this avoids alignment and strict-aliasing pitfalls and makes your code correct on both little- and big-endian hosts.

- *Can you explain what the `getaddrinfo()` function does and what it returns?*
  - `getaddrinfo()` is the modern, protocol-agnostic **name→address resolver**. You give it a *host* (e.g., `"example.com"` or `nullptr`) and a *service* (e.g., `"80"` or `"http"`), plus optional **hints** describing what kinds of sockets you want. It returns a **linked list** of results you can try in order (IPv6 and/or IPv4, stream or datagram, etc.).
  - **What it does**
    - Resolves the host name to one or more **network addresses** (DNS, /etc/hosts, or numeric literal).
    - Resolves the service to a **port number** (from `/etc/services` or numeric).
    - Applies your **hints** to filter/shape results (family, socket type, protocol, flags).
    - Produces ready-to-use `sockaddr` structs for `connect()` / `bind()`.
  - **Call shape**

```
int getaddrinfo(const char* host, const char* service, const struct
addrinfo* hints,struct addrinfo** res);
```

    - `host`: `"example.com"`, `"127.0.0.1"`, `"::1"`, or `nullptr` (see `AI_PASSIVE` below).
    - `service`: `"80"`, `"http"`, `"443"`, etc.
    - `hints`: optional filter (zero-initialize!).

- `res`: out-parameter → head of a linked list you must free with `freeaddrinfo()`.
- **What it returns (the list nodes)**

  Each node is a `struct addrinfo`:

```
    struct addrinfo {
    int ai_flags;                   // result flags
    int ai_family;                  // AF_INET, AF_INET6, AF_UNSPEC
    int ai_socktype;                // SOCK_STREAM, SOCK_DGRAM, ...
    int ai_protocol;                // IPPROTO_TCP, IPPROTO_UDP, ...
    socklen_t ai_addrlen;
    struct sockaddr* ai_addr;       // points to sockaddr_in /
sockaddr_in6
    char* ai_canonname;             // optional canonical name
    struct addrinfo* ai_next;       // next result
};
```

  You typically loop the list, try `socket(ai_family, ai_socktype, ai_protocol)`, then `connect()` (client) or `bind()` (server).
- **Common hints**

```
struct addrinfo hints{};
hints.ai_family   = AF_UNSPEC;      // both v4 and v6
hints.ai_socktype = SOCK_STREAM;    // TCP  (use SOCK_DGRAM for UDP)
hints.ai_flags    = 0;              // or AI_PASSIVE, AI_NUMERICHOST,
AI_ADDRCONFIG, ...
```

  - **Client connect:** `host="example.com"`, `service="443"`, no `AI_PASSIVE`.
  - **Server bind:** `host=nullptr`, `service="8080"`, set **`AI_PASSIVE`** so returned addresses are suitable for `bind()` (e.g., `INADDR_ANY` / `in6addr_any`).
  - **Other useful flags**:
    - `AI_NUMERICHOST` / `AI_NUMERICSERV` (skip name lookups),
    - `AI_ADDRCONFIG` (only families configured on the local host),
    - `AI_V4MAPPED` / `AI_ALL` (v4-mapped v6 behavior).

- *Can you explain what the linux functions `poll`, `epoll` and `select` do?*
  - `select`, `poll`, and `epoll` are Linux APIs for **I/O multiplexing**—letting one thread wait until **one or more** file descriptors (sockets, pipes, etc.) become readable, writable, or erroring, instead of blocking on just one. They all answer the same question ("which fds are ready?") but differ in **API shape**, **scalability**, and **features**. Without `poll` (or its cousins `select` / `epoll`), you'd have to block on one fd at a time, spawn a thread

per connection, busy-loop, or use signals—each with serious downsides (wasted CPU, complexity, poor scalability). `poll` enables **single-threaded event loops** and efficient servers: you hand the kernel a *set* of fds and react only when work exists.

- **`select`** is the oldest and most portable. You pass fixed-size bitsets (fd_sets) for read/write/except and a timeout. The kernel blocks until any fd becomes ready, then **modifies your bitsets in place** to mark the ready ones. Downsides: it scans linearly up to the **highest-numbered fd** every call (O(N)), so if you only care about fds **1** and **232** it will still iterate over **2…231** internally; it also copies bitsets in and out each time, and it has a historical **FD_SETSIZE** cap (often 1024) unless you recompile. It's fine for small fd counts or teaching code; it becomes clumsy and slow as N grows.
- **`poll`** removes the bitset cap. You pass an array of `pollfd` structs (fd + events mask), and the kernel returns with the `revents` set for each entry. Crucially, the **"highest-numbered fd" penalty from `select` is gone**— `poll` only inspects the descriptors **you list** (so watching fds 1 and 232 doesn't force scanning 2…231). It still does **O(N)** work per call over that array and still copies the whole array each time, but it handles arbitrary fd numbers and is simpler to manage than `select`. For a few hundred fds it's often enough; at thousands to tens of thousands, its per-call scan becomes the bottleneck.
- **`epoll`** (Linux-specific) is designed for **many** fds and event-driven servers. You create an epoll instance (`epoll_create1`), **register** fds once with `epoll_ctl(ADD/MOD/DEL)` , and then block in `epoll_wait` , which returns **only the fds that are ready** (no O(N) scan). Registration cost is amortized, so the per-wakeup cost is **O(#ready)**, which scales much better. `epoll` supports **level-triggered** (default: you're notified while the condition holds) and **edge-triggered** (ET: you're notified only on transitions; you must drain the fd until EAGAIN). In practice you use **non-blocking fds**, register for `EPOLLIN|EPOLLOUT|EPOLLET` as needed, and loop on read/write until EAGAIN inside the event handler.
- A few practical tips. With **`select`** / **`poll`** , readiness info is **one-shot**—you rebuild and resubmit the sets each call. With **`epoll`** , readiness state lives in the kernel; you add fds once and adjust with `epoll_ctl` . Always make sockets **non-blocking** to avoid stalls when multiple fds wake at once. Prefer **edge-triggered** only if you're comfortable writing "drain until EAGAIN" loops; otherwise, level-triggered is safer. For portability (BSD/macOS), the counterpart to epoll is **kqueue**; on Linux, epoll is the standard choice for high-concurrency servers.
- See this video for more info.

- *What is the difference between `poll` and `select` ?*

- `select` and `poll` both do I/O multiplexing (wait until one or more fds are ready), but they differ in API shape and scalability.
- **API & limits:** `select` uses fixed-size bitsets (`fd_set`) for read/write/except and takes `nfds = max_fd+1` ; it has a historical **FD_SETSIZE** limit (often 1024) unless you recompile. `poll` takes a **variable-length array** of `pollfd {fd, events, revents}` with **no hard fd number cap**, so it's friendlier to large or sparse fd sets.
- **Cost model:** both are **O(N) per call**—the kernel scans every fd you pass—but with `select` you also rebuild bitsets every time and it copies those in/out; with `poll` , you pass an array and the kernel fills `revents` in place. Neither scales well to tens of thousands of fds compared to `epoll` .
- **Semantics & ergonomics:** `select` **modifies** your bitsets (you must reinitialize them each call); `poll` leaves your `events` intact and sets per-entry `revents` . `select` reports "exceptional" conditions (often TCP OOB) via the except set; `poll` uses flags like `POLLPRI` . Timeouts differ in type— `select` uses `timeval` (µs), `poll` uses milliseconds. Both are **level-triggered**.
- **Practical upshot:** use `poll` over `select` when you might exceed FD_SETSIZE or want a cleaner per-fd mask; for high fd counts or best scalability on Linux, use **`epoll`** instead.
- See this thread and this thread for more info.

- *Can you explain how a message is typically transferred from the wire to a user process (mention the kernel internal buffer, userspace buffer, etc) in Linux? Mention also the typical number of copies that would be made.*
  - On a normal Linux system with a plain blocking TCP socket, the path from "wire" to your `recv()` buffer looks like this.
  - First, the network interface card (NIC) receives an Ethernet frame from the wire and, using DMA, writes the packet's bytes directly into host RAM. It doesn't write into your process memory; it writes into a kernel-managed receive ring buffer (descriptors + packet buffers) that the NIC driver set up beforehand. This DMA step is a "copy" in the physical sense (data moves from NIC to RAM), but it's not a CPU-driven `memcpy` , it's done by the NIC hardware.
  - Next, the kernel's network stack is notified that "there is a packet" via an interrupt or NAPI poll. The driver wraps the received data in a struct `sk_buff` (skb) that points to the DMA'd buffer. The skb is then passed up the stack: Ethernet layer checks MAC and type, IP layer checks the IP header, routing, fragmentation, etc., TCP layer checks sequence numbers, ACKs, and the TCP checksum. The payload is not copied again here; the skb just flows up with pointers and offsets adjusted. When the TCP stack accepts the segment as valid and in-window, it attaches the skb (or parts of it) to the

socket's receive queue, which lives entirely in kernel space and is limited by the socket receive buffer size (`SO_RCVBUF` , `/proc/sys/net/ipv4/tcp_rmem` , etc.). At this point, bytes are in kernel memory, queued for that socket, and your process might be woken up if it was sleeping in `recv()` or `select()/poll()/epoll_wait()` .

- Finally, when your user process calls `recv()` or `read()` on the TCP socket, the kernel copies data from the socket's receive queue into the user-provided buffer. Conceptually, the kernel walks through the skbs in the queue, copies up to the requested amount of payload bytes into your userspace address (using something like `copy_to_user()` which does access checks plus a `memcpy`), advances internal offsets, and frees or trims the skbs as data is consumed. This is the only place where a CPU-driven copy into userspace happens. After the copy, control returns to your process with the number of bytes delivered.
- In the simplest, typical case, the data path therefore has: one DMA transfer (NIC → kernel memory) plus one CPU copy (kernel socket buffer → user buffer). People usually talk about "one copy" for TCP receive on Linux, meaning one copy between kernel and user. There are more advanced zero-copy or "copy-avoidance" paths (e.g. `splice` , `io_uring` with registered buffers, some RDMA or DPDK setups) that try to skip that last copy, but for a normal TCP socket with `recv()` that's the standard story: wire → NIC → DMA into kernel buffer → TCP socket receive queue (still same memory) → `recv()` copies into your userspace buffer.
