

Topic 09

University Computer Science

What this course is, and how to use it

This is the ninth course in University Computer Science: **computer networks** – how a message crosses a planet by hopping machines that owe each other nothing, and what each layer promises the one above it.

Scope limit, stated once: a study course, not a degree; every header, checksum and sequence number in this handout is decoded byte by byte in the course suite, on the exact values printed here.

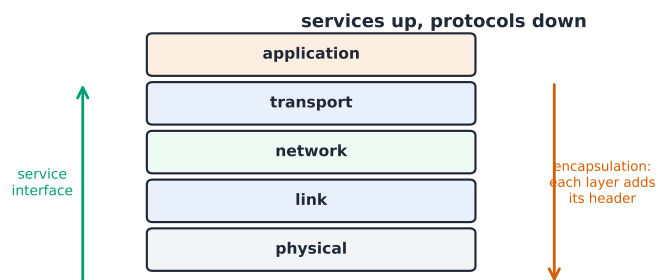
- Units 9.1–9.3 move a frame: layers, the link and IP, and routing.
- Units 9.4–9.5 make it reliable and useful: transport and application protocols.
- Units 9.6–9.7 secure and scale it: TLS and distributed fundamentals.

The network habit: **a protocol is a promise about bytes** – every unit decodes one promise at a time.

Layering and architecture

Why it matters. No single machine can implement the whole journey; **layering** lets each concern be built, debugged and replaced independently – the reason Wi-Fi could change without touching your browser.

The idea, with a picture. Each layer offers a **service** to the one above and consumes the one below: the five-course stack is **physical**, **link**, **network**, **transport**, **application**. Data is **encapsulated** downward (each layer adds its header) and decapsulated upward.



a 100-byte payload leaves the application as 54 bytes of headers around it: $14 + 20 + 20$

the hourglass: one narrow waist (IP), wide top, wide bottom

Each layer adds a header going down and strips it coming up; the service interface is the promise between neighbours

The derivation (why layers beat monoliths). A change confined to one layer requires no change elsewhere IF the service interface holds – the layer’s contract. Ethernet went from 10 Mb/s to 100 Gb/s under identical IP because the link service (best-effort frames) never changed. A leaky interface (an upper layer reading a lower header) destroys the independence: the browser that reads TTL is written once and breaks forever.

Worked example 1 – one message, twice wrapped. A 20-byte TCP segment rides a 20-byte IPv4 header inside a 14-byte Ethernet header: on the wire the frame is $14 + 20 + 20 = 54$ bytes of header for the payload – the receiver’s NIC strips 14, the IP layer strips 20, TCP strips its 20. Each layer sees only its own promise.

Worked example 2 – the reference models. The 5-layer course stack against the 7-layer OSI: OSI splits our link into physical + link and adds presentation + session above application. The internet won with the hourglass: narrow waist (IP), wide top (applications), wide bottom (links) – one protocol at the waist lets everything above talk to everything below.

Common mistakes. Counting layers in the wrong direction (services point UP the stack). Confusing the service (promise) with the protocol (the bytes that keep it). Skipping the physical layer in latency math (propagation is physics, not software).

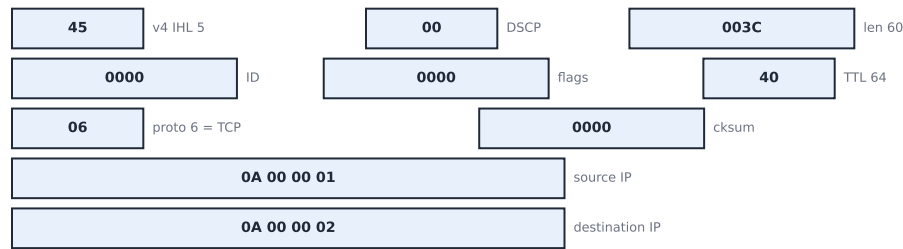
Summary. Services up, protocols down, encapsulation in between; the interface is the independence; IP is the waist of the hourglass.

Link layer and IP

Why it matters. The link moves a frame between neighbours; IP moves a packet between strangers. Confusing the two addresses is the classic beginner bug.

The idea, with a picture. An Ethernet frame: 6-byte destination **MAC address** MAC, 6-byte source, 2-byte type, payload, 4-byte **CRC**. An IPv4 header: version/IHL, total length, **TTL**, protocol, checksum, 4-byte source and destination **IP address** IP.

the course IPv4 header, 20 bytes in five rows



IHL 5 counts 32-bit words: $5 \times 4 = 20$ header bytes; total length 60 = 20 header + 40 payload
MAC names a neighbour; IP names the stranger

The IPv4 header decoded field by field on the course example: total length 60, TTL 64, protocol 6 (TCP)

The derivation (why IP checksums only the header). The CRC protects the whole frame link by link; the IP checksum protects only the header end to end – because routers rewrite the header (TTL decrements) but must not recompute data integrity. Each integrity mechanism is scoped to exactly the span its rewriter can touch.

Worked example 1 – decode a header. The course bytes 45 00 00 3C ... 40 06 ...: version 4, IHL 5 (so $5 \times 4 = 20$ bytes), total length $0x003C = 60$ (20 header + 40 payload), TTL $0x40 = 64$, protocol $0x06 = \text{TCP}$. The suite decodes every field of the printed header and asserts each value.

Worked example 2 – addresses at two scopes. The MAC changes every hop (frame to frame: it names neighbours); the IP never changes (end to end: it names strangers). A packet crossing 3 routers carries 4 different destination MACs in 4 frames and one destination IP the whole way – the suite traces all four frames.

Common mistakes. Reading IHL as bytes (it counts 32-bit words). Expecting the IP checksum to cover payload. Believing MAC addresses route (they find the next hop, chosen by IP).

Summary. MAC = neighbour, IP = stranger; IHL words; total length bytes; each checksum spans exactly what its rewriter touches.

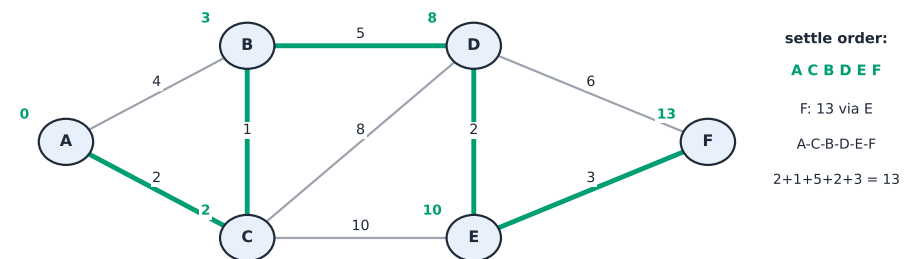
Routing

Why it matters. IP names the destination; routing decides the next hop – millions of machines agreeing without a boss, by an algorithm you can run on paper.

The idea, with a picture. A router's **routing table** maps destination prefixes to next hops: the **longest prefix match** rule picks the most specific route. Our course network: links with weights, routers A–F, and Dijkstra's

algorithm computing the shortest path tree from A.

weights and the shortest-path tree from A



longest prefix match picks the /24 over the /16 over the /8; the default catches the rest

The course graph with link weights; Dijkstra from A settles C, B, D, E, F in order with the tree in bold

The derivation (Dijkstra, in numbers). Course weights: A–B 4, A–C 2, B–C 1, B–D 5, C–D 8, C–E 10, D–E 2, E–F 3, D–F 6. From A, the settle order is C, B, D, E, F: C at 2 (via A), B at 3 (via C – the short hop through C beats A–B's direct 4), D at 8 (via B), E at 10 (via D – the two-hop $8 + 2$ beats the $2 + 10 = 12$ that C's direct link would give), F at 13 (via E: $10 + 3$ beats D's direct $8 + 6 = 14$). F's best path is A C B D E F at $2 + 1 + 5 + 2 + 3 = 13$. The suite runs the algorithm and asserts every settled distance, the order, and the tree.

Worked example 1 – longest prefix match. Table: $10.0.0.0/8 \rightarrow X$, $10.1.0.0/16 \rightarrow Y$, $10.1.2.0/24 \rightarrow Z$, default $\rightarrow W$. Destination $10.1.2.200$: three prefixes match; the /24 is longest – next hop Z. Destination $10.9.9.9$: only the /8 – X. The default catches what nothing else does.

Worked example 2 – why hierarchy. Flat tables would hold one entry per host: 4×10^9 IPv4 rows. Prefixes aggregate: an ISP announces one /12 covering 2^{20} addresses – the table stays at 10^5 entries because aggregation throws away the host bits.

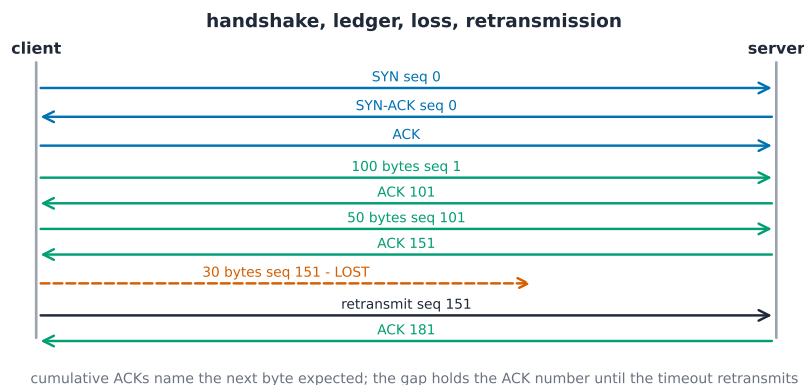
Common mistakes. Forgetting the settled set (relaxing an already final node). Matching the first table row instead of the longest prefix. Counting hops when the metric is delay.

Summary. Tables map prefixes to hops; longest prefix wins; Dijkstra builds shortest path trees by settling the nearest unsettled node; aggregation keeps the tables human-sized.

Transport (TCP/UDP)

Why it matters. IP promises best effort – packets may vanish, reorder, or duplicate. The transport layer connects application processes: TCP provides an ordered byte stream, while UDP sends datagrams without guaranteeing delivery or order.

The idea, with a picture. **UDP** : ports plus a length and checksum – a postcard. **TCP** : sequence numbers, acknowledgements, retransmission, flow and congestion control – a courier with a ledger. The **three-way handshake** SYN, SYN-ACK, ACK opens a connection; FIN closes it.



The course exchange: handshake, two segments with cumulative ACKs, a lost segment retransmitted after the timeout

The derivation (why cumulative ACKs). Each ACK names the next byte expected: ACK 200 says bytes 0–199 arrived, whatever their order. A gap stalls the ACK number – the sender’s timeout retransmits from the gap. One number summarises the whole receive state: simple to check, impossible to misread.

Worked example 1 – the ledger. SYN consumes sequence 0; the server SYN-ACK carries its own 0. Client sends 100 bytes: seq 1, next 101; server ACKs 101. Client sends 50: seq 101; server ACKs 151. A 30-byte segment sent at seq 151 is lost: the ACK stays 151 until the timeout retransmits it – then ACK 181. The suite simulates the exchange and asserts every number.

Worked example 2 – UDP’s honest scope. A 8-byte UDP header (ports, length, checksum) against TCP’s 20 minimum: DNS fits in one UDP datagram – a request is one packet, and if it is lost the client re-asks. Video calls tolerate loss but not delay: retransmitting a frame due in 30 ms arrives too late to matter – UDP plus application logic is the right promise.

Common mistakes. Counting SYN in the data sequence (it consumes one number but carries none). Confusing flow control (receiver’s window) with congestion control

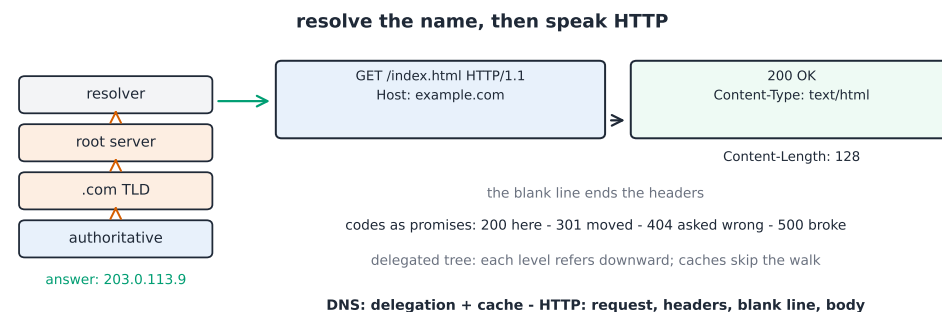
(network’s state). Expecting TCP to guarantee timing – it guarantees order and delivery, never latency.

Summary. UDP = ports plus a checksum for request-sized work; TCP = sequence ledger plus cumulative ACKs plus retransmission; the handshake opens, FINs close, and windows protect both ends.

Application protocols

Why it matters. The transport moves bytes; the application agrees on what they mean. Reading a protocol is reading a conversation with a grammar.

The idea, with a picture. **DNS** : name → address, hierarchical delegation (root → TLD → authoritative). **HTTP** : request line, headers, body; **status codes** 2xx success, 3xx redirect, 4xx client error, 5xx server error.



A DNS lookup then an HTTP exchange: the recursive resolver asks root, TLD, authoritative; the browser then GETs and the server answers 200

The derivation (why DNS is hierarchical). One global name table would be one global outage and one global fight. Delegation splits it: the root (13 addresses) knows the TLD servers; .com servers know the registrars; the authoritative server knows the host. Each answer is a referral downward – and caching turns the tree into a short-cut: the resolver that once asked needs no full walk for the TTL of the answer.

Worked example 1 – one page, by the bytes. GET /index.html HTTP/1.1 + Host: header; the server answers 200 OK, headers (Content-Type: text/html, Content-Length: 128), blank line, 128 bytes. The suite checks: a response with Content-Length 128 carries exactly 128 body bytes; a 404 carries headers and a short body; a 301 carries Location: and no body to speak of.

Worked example 2 – the codes, read as promises. 200 (here is the body), 301 (the body moved, Location says where), 404 (you asked wrong), 500 (I broke). A crawler treats 301 as a pointer to follow, 404 as a dead end, 500 as retry-later – the code is the contract the client programs against.

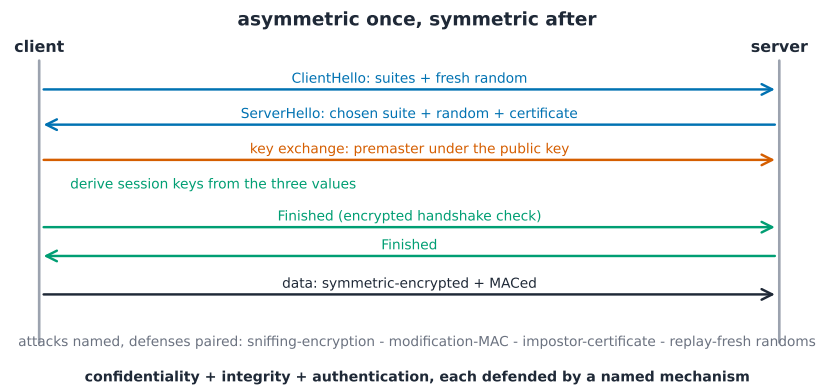
Common mistakes. Forgetting the blank line that ends headers (the parser waits forever). Reading Content-Length as header bytes. Treating DNS as one server (it is a delegated tree with caches).

Summary. DNS resolves names by delegation plus caches; HTTP is a request-response grammar where the status code is the promise and headers carry the meta-data.

Network security

Why it matters. The network is a party line: anyone on the path can read, drop, or forge. Security is the set of promises that make an open channel safe to use.

The idea, with a picture. **Symmetric** encryption: one shared key, fast. **Asymmetric** : a key pair – public encrypts, private decrypts. **TLS** combines them: asymmetric for the key exchange, symmetric for the data, certificates for identity.



ClientHello, ServerHello with certificate, key exchange, both Finished – then symmetric data

The derivation (the properties, and who provides what). Confidentiality needs encryption; integrity needs a **MAC** with both directions checked (each Finished message proves the handshake was not tampered with); authentication needs a certificate chain to a root the client already trusts. Drop any one and the protocol fails differently: no encryption is a wiretap, no MAC is alteration, no certificate is an impostor.

Worked example 1 – the handshake, numbered. ClientHello (offered suites, random), ServerHello (chosen suite, random) + certificate, key exchange (the pre-master encrypted to the server’s public key), both sides derive the session keys from the three values, Finished messages verify, then data flows symmetric-encrypted. One RSA operation, then stream-speed cryptography – the asymmetry is used only to agree on the symmetric key.

Worked example 2 – the attacker menu. Passive sniffing: defeated by encryption. Modification: defeated by the MAC (the handshake Finished catches tampering before data flows). Impersonation: defeated by certificates (the server proves possession of the private key matching the certified public one). Replay of the handshake: defeated by the fresh randoms in both Hellos. Each attack names its defense.

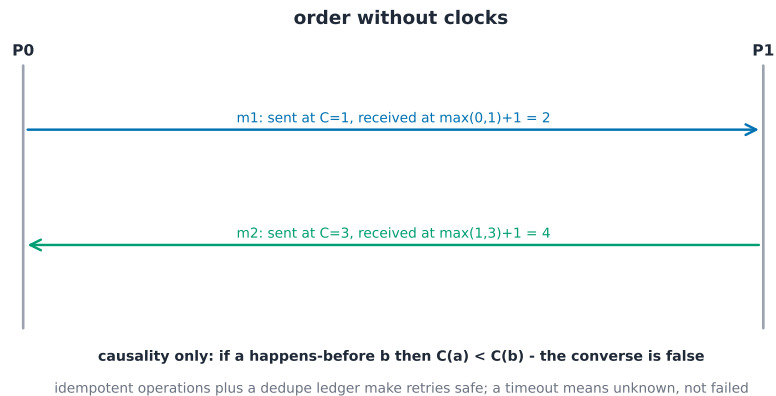
Common mistakes. Encrypting without authenticating (you have a secure channel to the wrong machine). Trusting self-signed certificates in production. Believing HTTPS makes the server honest (it makes the channel private and the server identified – the content can still lie).

Summary. Symmetric for speed, asymmetric for key agreement, certificates for identity, MACs for integrity; every defense answers a named attack.

Distributed systems fundamentals

Why it matters. Once two machines cooperate over a network, partial failure and reordering become physics. Distributed systems is the discipline of promising things anyway.

The idea, with a picture. A **distributed system** shares state across machines with **partial failure** : one machine crashing must not corrupt the rest. Messages reorder and duplicate; **idempotence** makes re-execution safe; **happens-before** orders events even without clocks.



Two messages, one causal chain: Lamport counters 1, 2, 3, 4 across P0 and P1 without any clock

The derivation (why clocks cannot order). Two machines' clocks skew by unbounded amounts, so timestamps cannot safely order events across machines. The **Lamport clock** Lamport gives a counter that respects causality: on send, attach $C+1$; on receive, set $C = \max(C, \text{received}) + 1$. If a happens-before b then $C(a) < C(b)$ – the converse does not hold. A smaller counter alone does not prove a causal relationship between two events.

Worked example 1 – the Lamport trace. P0 sends m_1 at $C = 1$; P1 receives: $C = \max(0, 1) + 1 = 2$; P1 sends m_2 at 3; P0 receives: $C = \max(1, 3) + 1 = 4$. The suite asserts every counter and that $C(m_1) = 1 < C(m_2) = 3$ – causal order preserved without any clock.

Worked example 2 – idempotence by design. A payment API takes `deduct(user, 25, id=xyz)`: executing it twice must deduct once. Make the operation idempotent by key – the ledger records `xyz` as done and the retry is a no-op. Retry-safe systems are built from idempotent operations plus a dedupe ledger, not from hoping the network stops duplicating.

Common mistakes. Using timestamps for cross-machine order (skew wins). Retrying non-idempotent operations (double charges). Assuming a call that returned means the work happened – or that a timeout means it did not (it means you do not know: the third outcome).

Summary. Partial failures may be temporary or permanent; messages reorder and duplicate; Lamport counters respect happens-before; idempotence plus dedupe makes retries safe; a timeout means unknown, not failed.