CS-202 Computer Systems
Summary

Isaac Metthez

Internet Architecture

Internet Layers (top to bottom)

1. **Application** – processes (browser, DNS client…)
2. **Transport** – TCP, UDP
3. **Network** – IP
4. **Link** – WiFi, Ethernet, FTTH…
5. **Physical** – electrical / EM / optical signals

Encapsulation: each layer adds its own header.
Message → TCP segment / UDP datagram → IP packet → Link frame.

Same-layer (cross-device) = communication **protocol** (messages in headers).
Cross-layer (same device) = API / syscall interface.

Computers implement all 5 layers.
Packet switches implement only the lowest 2–3.
Each layer reads/interprets *only* its own header.

I/O Stack (implementation view)

Processes → Syscall interface → FS / Network subsystem → Device drivers → Devices

- **Internet stack** = goal of functionality (what layer).
- **I/O stack** = how it is implemented (OS-level).
- FS is **device-independent**; drivers are **device-specific**.
- Block interface between FS and device drivers.
- Need: ≥ 1 FS, as many drivers as device types.

Why Layers & Interfaces?

Abstraction → separation of concerns → **simplicity** + **flexibility**.
Does *not* directly improve performance.

Internet Structure & ISP Hierarchy

The Internet Edge

- End-systems connect to the Internet via **access ISPs** (Swisscom, Sunrise…).
- Connectivity technologies: cellular towers, cable/DSL/fiber to the home, satellite.
- University/enterprise networks connect to ISP **PoPs** (Points of Presence).

ISP Hierarchy (3 tiers)

- **Tier-3 / Access ISPs**: connect end-users directly. Customers of tier-2.
- **Tier-2 / Regional ISPs**: connect tier-3 ISPs. Customers of tier-1.
- **Tier-1 ISPs**: global backbone, nobody's customer. All tier-1s are directly connected.

Economic relationships:

- **Customer–provider**: lower tier pays higher tier for connectivity.
- **Peering**: two ISPs at the same level agree to exchange traffic for free (e.g. two tier-2 ISPs in the same country whose customers exchange a lot of traffic).

IXPs – Internet Exchange Points

As ISPs formed many peering relationships, it became cheaper to connect to a central **IXP** (giant switchboard) rather than draw a separate cable to every peer.

Content Providers & Edge Caching

- Large content providers (Google, Amazon…) grew from simple servers to owning their own network infrastructure ("clouds").
- **Edge caching**: cloud installs cache servers *inside* an ISP's network, owned and managed by the cloud.
- Benefits: (1) reduces traffic between ISP and cloud → lower ISP costs; (2) improves performance for ISP's end-users.

Processes & Addressing

Process Identification

Global process name = (IP address, port number)

- **IP address**: identifies the end-system.
- **Port number**: identifies the process on that end-system.
- Well-known ports: HTTP = 80/8080, HTTPS = 443/8443, DNS = 53.
- Client ports: assigned automatically by OS.

Client-Server Model

- **Server**: always-on, well-known address, listens for requests.
- **Client**: initiates contact, intermittent.
- First design decision: how many processes, what role, what protocol.

Network Topology

- End-systems connected via **packet switches** (routers, L2 switches).
- Connected by **links** (copper, fiber, wireless).
- Organized in a **hierarchy** → simplifies & reduces cost.
- ISP PoP (Point of Presence) connects to Internet.

The Web & HTTP

HTTP Protocol

HyperText Transfer Protocol – request/response between web client & web server.
Uses **TCP** (port 80 / 443).

Requests:

- **GET** `<URL>` – request a web object
- **HEAD** `<URL>` – metadata only (no data body)
- **POST** `<URL>` – send data to server (forms)

Responses:

- **200 OK** – here is the object
- **301 Moved Permanently** – new location provided
- **304 Not Modified** – object unchanged (conditional GET response)
- **400 Bad Request** – malformed request
- **404 Not Found**

Web Page Loading

User types URL → GET base file → 200 OK → parse base file, find referenced URLs → GET each one → multiple request/response pairs.

Stateless vs. Stateful

- HTTP is **stateless**: server forgets past exchanges.
- **Cookies** add state: server sends cookie in response → client stores it → client attaches cookie to future requests to same domain → server reads cookie to personalize.

Web Caching (Proxy Server)

- Proxy sits between clients and origin server.
- If cached → immediate response; else fetch, cache, respond.
- Reduces client latency + server load.
- Origin server sets **max-age** in response header.
- After expiry, proxy uses **conditional GET**:
  - Proxy sends: `GET <URL>` with `If-Modified-Since: <date,time>`
  - If unchanged: server replies **304 Not Modified** (header only, no data body) → proxy serves cached copy.
  - If changed: server replies **200 OK** with new data.

Edge Caching (vs. Proxy Caching)

- **Proxy cache**: separate entity between client and origin server.
- **Edge cache**: cloud (Google, Amazon…) installs cache servers *inside* the ISP's network. Owned/managed by the cloud. Stores most popular objects for that ISP's users.

DNS – Domain Name System

Purpose

Translates DNS names (e.g. `www.youtube.com`) → IP addresses. Port **53**, uses **UDP**.

DNS Hierarchy

- **Root**: knows IP of all TLD servers.
- **TLD** (.com, .ch…): knows auth. servers for their TLD.
- **Authoritative**: knows all IPs for its domain (e.g. *.youtube.com).
- **Local DNS / resolver**: every client knows ≥ 1.

Hierarchy is a *universal technique for scalability*: each node keeps state only about its children.

Resolution

Client → local DNS → root → TLD → authoritative → response back.

Recursion: response flows back on same path.
Iteration: server gives hint ("try this server").
Typical: local DNS = recursive; others = iterative.

DNS Caching & TTL

Each response includes **TTL**. Servers cache for that duration.
TTL decreases as response propagates through hierarchy.
No conditional GET for DNS: responses are tiny (just an IP address), so a dynamic freshness check would not save significant data compared to just re-fetching.

Security: DDoS

Flood root/TLD with queries → cannot serve legitimate requests.

End-to-End: Typing a URL

Processes Involved

On your computer: web browser (client), DNS resolver (stub client).
On remote computers: local DNS server, root DNS server, TLD DNS server, authoritative DNS server, web server (origin or proxy/edge cache).

Step-by-step

1. Browser extracts hostname from URL (e.g. `www.youtube.com`).
2. Browser asks local DNS resolver to resolve the hostname.
3. DNS resolution (may involve root → TLD → authoritative), using **UDP** on port 53. Returns IP address.
4. Browser opens **TCP connection** to (IP, port 80/443): 3-way handshake.
5. Browser sends **HTTP GET** request for the base HTML file.
6. Web server responds with **200 OK** + HTML data.
7. Browser parses HTML, finds embedded objects (images, CSS, JS…).
8. Browser sends additional **HTTP GET** requests for each embedded object.
9. Each object → separate request/response pair.

Role of Caching (which messages can be avoided?)

- **DNS caching** (at local DNS resolver or client): if hostname was recently resolved and TTL has not expired → skip root/TLD/auth queries entirely.
- **Web proxy cache**: if object is cached and not expired → proxy responds directly, no request to origin server.
- **Conditional GET**: if cached object has expired → proxy sends conditional GET → if unchanged, avoids transferring the data body.
- **Edge cache**: cloud cache inside ISP network → avoids crossing the wider Internet to reach the origin server.
- **Browser cache**: browser itself may cache objects locally.

Network & File Syscalls

Connectionless (UDP-style)

```
// Server                          // Client
s = socket(...)                    sock = socket(...)
bind(s, [IP,53],...)               // auto port
recvfrom(s, buf,..., &src)         sendto(sock, data,..., [IP,53])
sendto(s, resp,..., [IP,port])     recvfrom(sock, buf,...)
```

`sendto`/`recvfrom`: specify remote each time. No connection.

Connection-Oriented (TCP-style)

```
// Server                          // Client
s1 = socket(...)                   sock = socket(...)
bind(s1, [IP,443],...)
listen(s1,...)                     connect(sock,[IP,443],...)
s2 = accept(s1,...)                // connect returns
send(s2, data,...)                 recv(sock, buf,...)
recv(s2, buf,...)                  send(sock, data,...)
close(s2); close(s1)               close(sock)
```

- **Listening socket** (s1): receives connection requests.
- **Connection socket** (s2): data exchange with specific client.
- `send`/`recv`: no remote needed – socket is connected.

Conceptual Socket Questions (exam-relevant)

- A TCP server needs **1 listening socket** + **1 connection socket per active client**.
- A UDP server needs only **1 socket** (no connection state).
- A TCP client needs **1 socket** (it calls `connect`).
- The listening socket and connection sockets have *different* file descriptors.

File Syscalls

```
fd = open('/path', O_RDWR);
fd = open('/path', O_RDWR|O_CREAT, perms);
read(fd, buf, count);    write(fd, buf, count);
lseek(fd, offset, whence);    close(fd);
```

- **File descriptor** = abstraction (file as byte sequence).
- Each fd has an **offset** (next read/write position).
- Flags: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_CREAT`.

File Permissions

`drwxr-xr-x` = type + user/group/others (rwx).
Each file has **UID** and **GID**.

Network Performance

Metrics

- **Packet loss**: fraction of packets lost (%).
- **Packet delay**: time src → dst (e.g. 10 ms).
- **Throughput**: data rate (bits/sec).

Delay Components

$$\text{Packet delay} = \underbrace{\frac{L}{R}}_{\text{transmission}} + \underbrace{\frac{d}{s}}_{\text{propagation}} + t_{\text{queue}} + t_{\text{proc}}$$

- **Transmission** = L/R (pkt size / link rate). Depends on pkt + link.
- **Propagation** = d/s (length / speed). Depends on link only.
- **Queuing**: depends on other traffic. → ∞ if avg arrival rate $> R$.
- **Processing**: switch processes pkt (usually small).
- Max queue delay (capacity N+1 pkts): $N \cdot L/R$.

With Store-and-Forward Switch

delay = $L/R_1 + d_1/s_1 + t_q + t_p + L/R_2 + d_2/s_2$
Store&forward: wait for *all* bits before forwarding.
Cut-through: forward after reading headers (not default in course).

Throughput

File F bits, packets L bits, link rate R:
Transfer time $\approx F/R$ (propagation usually negligible).
Avg throughput $\approx R$ (always slightly $< R$).
Multiple links: throughput $\approx \min\{R_1, R_2, \ldots\}$.
Bottleneck link = slowest link on the path.

With a store&forward switch (2nd link rate $R' > R$):
Transfer time = $F/R + L/R' +$ prop. delays. Avg throughput $\approx \min\{R, R'\} = R$.
Adding more switches does not change this: throughput $\approx$ rate of the bottleneck link.

Delay vs. Throughput

Delay: small interactive messages (gaming, voice).
Throughput: bulk transfers (downloads).
Related but not in an obvious way.

Switching

Packet Switch Internals

- Multiple links (bidirectional), queues (1 per outgoing link), forwarding table.
- Arrival: read headers → forwarding table → choose link → enqueue.
- Queue drains at link rate. Queue full → packet **dropped**.

Packet vs. Circuit Switching

Packet Switching	Circuit Switching
Packets on demand	Resources reserved in advance
Per-packet decisions	Per virtual-circuit
Efficient: idle resources shared	**Predictable**: guaranteed
Unpredictable perf.	No surprise loss/delay
Simpler switches	Complex switch logic
Internet uses this	*Old telephone networks*

Key tradeoff: packet switching is more efficient (serves more clients) but performance is unpredictable. Circuit switching guarantees perf. but wastes idle resources.

Example

10 clients, 10 Gbps link, 1 active, downloading 10 Gbit:
Circuit (1 Gbps each): **10 s**. Packet (all resources): **1 s**.

Network Congestion

Traffic > capacity ⇒

- **Packet loss**: dropped at full queues.
- **Queuing delay**: packets wait.

UDP vs. TCP

UDP	TCP
Connectionless	Connection-oriented (3-way handshake)
Unreliable (no retransmission)	Reliable (checksums, SEQ/ACK, timeouts)
No flow control	Flow control (`rwnd`)
No congestion control	Congestion control (`cwnd`)
Lightweight, low overhead	Higher overhead
Mux/demux only	Mux/demux + reliability + flow + congestion
`sendto`/`recvfrom` (specify remote)	`send`/`recv` (connected socket)

Use cases:

- **UDP**: DNS (port 53) – small messages, low latency, loss tolerable.
- **TCP**: HTTP/HTTPS (port 80/443) – reliability needed for web pages.

Both provide: multiplexing (sender labels segments with src/dst port) and demultiplexing (receiver identifies destination process and delivers message).

Transport Layer & TCP

Transport Services

- **Multiplexing**: sender creates segments with src/dst port #.
- **Demultiplexing**: receiver identifies dst process, delivers msg.

TCP 3-Way Handshake

1. Client → Server: **SYN**
2. Server → Client: **SYN-ACK**
3. Client → Server: **ACK** (can include data)

TCP Reliability

- **Checksums**: detect corruption.
- **SEQ** (Sequence #): # of first data byte. Receiver detects retransmission vs. new.
- **ACK** (Acknowledgment #): next expected byte. Neg ACK → retransmit.
- **Timeouts**: detect loss → retransmit.

Flow Control vs. Congestion Control

- **Flow control**: don't overwhelm *receiver*. **rwnd** set by receiver (TCP header).
- **Congestion control**: don't overwhelm *network*. **cwnd** estimated by sender.
- Sender window = min(rwnd, cwnd)

TCP Congestion Control

TCP Tahoe – 2 states

Slow Start (exponential):

- Start: cwnd = 1 MSS.
- New ACK ⇒ cwnd += 1 MSS. Window **doubles** each RTT.
- cwnd ≥ ssthresh ⇒ Congestion Avoidance.

Congestion Avoidance (linear):

- New ACK ⇒ cwnd += MSS^2/cwnd. Grows **+1 MSS / RTT**.

On Timeout: ssthresh ← cwnd/2, cwnd ← 1 MSS, retransmit, → Slow Start.

TCP Reno – 3 states (adds Fast Recovery)

On 3 duplicate ACKs:

- ssthresh ← cwnd/2.
- cwnd ← **ssthresh** (not 1 MSS!).
- Retransmit oldest unACKed. → **Fast Recovery**.

Fast Recovery:

- Dup ACK ⇒ cwnd += 1 MSS (inflate).
- New ACK ⇒ cwnd ← ssthresh, → Congestion Avoidance.

On Timeout: same as Tahoe (cwnd → 1 MSS, → Slow Start).

Tahoe vs. Reno – Key Difference

Event	Tahoe	Reno
Timeout	cwnd → 1 MSS, SS	cwnd → 1 MSS, SS
3 dup ACKs	cwnd → 1 MSS, SS	cwnd → ssthresh, FR

Reno is less aggressive on dup ACKs: halves instead of resetting.

Peer-to-Peer (P2P) File Sharing

P2P vs. Client-Server

- **No clear separation of roles**: peer processes generate both requests and responses.
- Typically **no dedicated infrastructure** – runs on end-users' computers.
- Decentralized system: no single authority, no single point of trust.

How P2P File Sharing Works

1. Content is split into **chunks** (numbered pieces).
2. A peer wanting to download a file needs to find which peers have it.
3. The peer queries a **directory service** (see below) with the content's metadata or global content ID.
4. The directory returns a **list of peers** (with their IP:port) that store the content.
5. The peer connects directly to those peers and downloads different chunks from different peers in parallel.

Directory Service: Tracker vs. DHT

Both provide the same service: **input** = global content ID → **output** = list of peers storing that content.

- **Tracker**: centralized server that maintains the directory. Simpler but single point of failure.
- **DHT** (Distributed Hash Table): the directory is itself distributed across peers. Fully decentralized.
- You don't need both – they are different implementations of the same service.