☑ Application Layer (Layer 5 in the Internet Stack)

The **application layer** is the **topmost layer** of the Internet protocol stack. This layer is **closest to the end user** and is responsible for **network applications**—programs that run on different end systems and **communicate over the network**.

🔹 Main Functions:

- Enables **process-to-process communication** between applications on different hosts.
- Provides **network services** to user applications (like web browsers, email clients, etc.).
- Uses protocols to **define how applications communicate**.

🔹 Common Application Layer Protocols:

Protocol	Purpose
HTTP (HyperText Transfer Protocol)	Used by web browsers to access web pages.
SMTP (Simple Mail Transfer Protocol)	Sends emails.
POP3/IMAP	Retrieves emails from the mail server.
DNS (Domain Name System)	Translates human-readable domain names to IP addresses.
FTP (File Transfer Protocol)	Transfers files between computers.

🔹 How It Works:

- A user runs a **client application** (like a browser).
- The application uses a **specific protocol** (e.g., HTTP) to request a service.
- The server responds using the same protocol.

For example:

- When you enter a URL, your browser (client) sends an HTTP request.
- A web server receives the request and replies with an HTTP response containing the web page.

🧠 Key Concepts:

- **Client-Server Model:** The client initiates the communication; the server waits and responds.
- **Peer-to-Peer Model (P2P):** Devices (peers) act as both client and server (e.g., BitTorrent).
- **Message Format & Syntax:** Every protocol defines how messages are structured and understood.
- **Port Numbers:** Help identify which application to deliver data to (e.g., port 80 for HTTP, 25 for SMTP).

🔚 Summary:

- The **Application Layer** is where **network services** like web browsing, email, file transfer, and DNS happen.
- It provides **communication services to applications** and relies on **lower layers** (like transport and network) to send data over the internet.

The **application layer** is the **only layer** in the network stack that the user **directly interacts with** through software applications such as:

- Web browsers (Chrome, Firefox) – use **HTTP**
- Email clients (Outlook, Gmail) – use **SMTP, IMAP, POP3**
- Chat apps (WhatsApp, Messenger) – use various application-layer protocols
- File transfer apps (FileZilla, browser downloaders) – use **FTP, HTTP**

🧠 Key Point:

Other layers (transport, network, link, physical) are responsible for moving data behind the scenes and are **not visible** to the user.

The work of application layer is:

The application layer is responsible for facilitating the data that users want to access, transmit, or receive through the network. It implements the services needed for communication between software applications on different devices.

📘 Application Layer and Transport Layer Relationship

The **application layer** takes service from the **transport layer** to send and receive data between processes running on different end systems.

- The **application layer** is responsible for user-level services (like web browsing, email, file transfer).
- But it **does not directly manage data transmission**—it relies on the **transport layer** for that.

🔄 How It Works:

- The application layer creates a message (e.g., an HTTP request).
- It passes this message **down to the transport layer**.
- The transport layer (using **TCP** or **UDP**) adds headers and **ensures reliable or fast delivery**, depending on the protocol.
- After that, lower layers (network, link, physical) handle actual data delivery.

🧠 Example:

When you browse a website:

- Your browser (application layer) creates an HTTP request.
- It is passed to the transport layer, which wraps it in TCP.
- The TCP segment is sent to the destination via the lower layers.

☑ Final Line :

The application layer depends on the transport layer to deliver its messages to the correct application process on the destination host.

Application Layer Services: Two Main Models

The **Application Layer** provides services through two main models:

---

1 Client-Server Model

- The **client** requests services.
- The **server** is always-on and provides services.
- Clients **do not communicate with each other**, only with the server.

🟢 **Examples:**

- Web browsing (HTTP)
- Email (SMTP, IMAP)
- Online banking

2️⃣ Peer-to-Peer (P2P) Model

- **All devices (peers)** act as both **clients and servers**.
- They **share resources** directly with each other.
- No need for a central server.

🟢 **Examples:**

- File sharing (BitTorrent)
- Skype
- Blockchain nodes

🧠 Summary Line:

The application layer supports **Client-Server** and **Peer-to-Peer** models to allow process-to-process communication over the internet.

All network applications run on **hosts** (end systems), which communicate over the Internet using client-server or peer-to-peer models.

Base stations only provide access to the network. They **do not run application-layer software** or services.

Base stations, gateways, routers, and switches are part of the **network infrastructure**, and their main job is to **move data**, **not consume or produce** it at the application layer.

Base stations, gateways, routers, and switches are producers or forwarders of information, not consumers. They provide access and delivery but do not run or interact with application-layer services.

Applications on different hosts communicate over the network using a layered architecture. The network enables this communication by transferring data from one host to another.

In an **end system** (like a laptop, smartphone, or server), **all five layers of the Internet protocol stack** are present and active:

🔹 The 5 Layers:

1. **Application Layer** – Runs the user's application (e.g., browser, email, chat app).
2. **Transport Layer** – Ensures reliable or fast data transfer (e.g., TCP, UDP).
3. **Network Layer** – Routes the data across the Internet (e.g., IP).
4. **Link Layer** – Handles data transfer between directly connected devices (e.g., Ethernet, WiFi).
5. **Physical Layer** – Transmits raw bits over the physical medium (cable, radio, etc.).

🧠 Key Concept:

The **end system** uses all 5 layers to **send and receive data** from one host to another across the network.

☑ Final Note:

Only end systems (not intermediate devices like routers or switches) **implement the full five-layer Internet protocol stack**.

🌐 Layer Usage in Core Network Devices

Unlike end systems, **core network devices** (such as routers, switches, and hubs) **do not use all five layers** of the Internet protocol stack because they **do not run application-layer programs**.

📊 Layer-wise Role of Core Devices:

Device	Layers Used	Reason
Router	**Network, Data Link, Physical**	For routing packets across networks
Switch	**Data Link, Physical**	For forwarding frames within a LAN
Hub	**Physical**	Only repeats electrical signals

🧠 Key Concept:

Core devices **do not run the application or transport layers** because they are not end systems. Their job is to **move data**, not create or use it.

☑ Final Note:

Core devices use only the layers they need to perform their specific roles in the network. They forward or switch data, not process it like end systems.

Types of Application Architectures

There are **two main types** of application architectures in networking:

---

1 Client-Server Architecture

- In this model, there is a **centralized server** that provides services or data.
- **Clients** initiate requests, and the **server responds**.
- The server is usually **always-on** and can handle multiple clients at once.

☑ **Examples:** Web (HTTP), Email (SMTP/IMAP), Cloud services

---

2 Peer-to-Peer (P2P) Architecture

- In P2P, there is **no central server**.
- Each device (peer) can act as **both a client and a server**.
- Peers **directly communicate** and share resources with each other.

☑ **Examples:** BitTorrent, Skype (early versions), Blockchain

🌐 Application Architectures: Client-Server vs Peer-to-Peer

🖥️ Client-Server Paradigm

In the **client-server model**, there is a **central server** that provides services, and multiple clients connect to it.

◇ *Server:*

- Always-on host
- Has a **permanent IP address**
- Usually hosted in **data centers** for high availability and scalability

◇ *Clients:*

- **Initiate contact** with the server
- May have **dynamic IPs**
- Can be **intermittently connected**
- **Do not communicate with each other directly**

☑ Examples:

- HTTP (Web)
- FTP (File Transfer)
- IMAP (Email)

🔄 Peer-to-Peer (P2P) Architecture

In the **P2P model**, there is **no always-on server**. All end systems (called peers) can act as both client and server.

◇ *Characteristics:*

- Peers **directly communicate** with each other
- Each peer can **request** and **provide** services
- **Self-scalable** – more peers = more service capacity
- Peers may be **intermittently connected**
- Peers may have **changing IP addresses**
- **Management is complex** due to decentralization

☑ Examples:

- BitTorrent
- Older versions of Skype
- P2P file sharing networks

🧠 Summary Line for Notes:

The **Client-Server model** uses a central server for all communication, while **P2P architecture** allows direct interaction between peers, making it more scalable but harder to manage

☁️ Cloud Computing and Data Centers in Modern Networking

Nowadays, **big companies** are building large **data centers** and offering **cloud computing services**. These data centers are connected to **powerful servers** that are always online.

Through these connections:

- We maintain **centralized servers**.
- These servers allow **simultaneous access** to data from **any location in the world** via the Internet.
- Users can store, retrieve, and manage data **anytime, anywhere**.

Nowadays big companies are creating data centers and cloud computing services and connecting them to servers. Through this communication, we maintain a server that can be accessed simultaneously from anywhere.

Clients connect and communicate with the server **intermittently**. Whenever needed, they turn on and access the server to retrieve data or information. **Clients do not have fixed (static) IP addresses**; instead, they usually use **dynamic IPs** assigned by the network.

🧠 Notes:

- **Clients** are **not always online**. They only connect when they need to access services (like web browsing or downloading files).
- Most clients use **Dynamic IP Addresses**, assigned by **DHCP** (Dynamic Host Configuration Protocol).
- In contrast, **servers** usually have **static IPs**, as they need to be reachable at all times.

🌐 Private IP vs Public IP

🔒 Private IP:

- Used inside a local network (home, office).
- Not accessible from the Internet.

- Assigned by router.
- Example: `192.168.0.1`

🌎 Public IP:

- Used on the Internet.
- Globally unique and reachable.
- Assigned by ISP.
- Example: `8.8.8.8` (Google DNS)

➡️ **Private IP = Local use**
➡️ **Public IP = Internet use**

📡 Common Application Layer Protocols and Their Functions:

🔹 **HTTP (HyperText Transfer Protocol):**
Used for **transferring web pages** (text, images, etc.) between **web browser (client)** and **web server**.
 Example: When you open a website, HTTP gets the page from the server.

🔹 **FTP (File Transfer Protocol):**
Used for **uploading and downloading files** between a **client** and a **server**.
 Supports both directions: client → server, server → client.

🔹 **SMTP (Simple Mail Transfer Protocol):**
Used to **send emails** from a client to a mail server or between mail servers.
 Example: Your email app uses SMTP to send your message out.

🔹 **IMAP (Internet Message Access Protocol):**
Used to **retrieve emails** from a mail server, while keeping a copy on the server.
 Allows accessing your email from **multiple devices**.

☑ Summary Table:

Protocol	Purpose	Direction
HTTP	Load websites	Browser ↔ Web Server
FTP	Transfer files	Client ↔ Server
SMTP	Send email	Client → Server
IMAP	Read email (from server)	Server → Client

The operations on the **network backbone** are handled by **network engineers**.
End users do not directly interact or work at this level.

📚 Note-style Explanation:

- The **network backbone** is the **core part of the Internet**, made up of high-speed, interconnected routers and links.
- It ensures **fast and reliable data transmission** across long distances.
- This layer is **managed by professionals** (like ISPs, data center engineers, and network admins).
- **Users** only interact with the **edge of the network** (e.g., apps, websites) — **not with the core infrastructure**.

📘 Program (Definition):

A **program** is a **set of instructions** written in a programming language that performs a specific task when executed.
It is a **static** and **passive** entity, stored on disk (e.g., `.exe`, `.py`, `.java` files).
It does not perform actions on its own until it is run.

📝 Example:
A file named `music_player.exe` stored on your computer is just a program — not running yet.

🖥️ Process (Definition):

A **process** is a **program in execution**.
It is an **active** entity, created when a program is run.
A process includes the program code, **program counter**, **CPU registers**, **memory space**, **open files**, and **resources** it needs to execute.

📝 Example:
When you double-click `music_player.exe`, a **process** is created to play music — now it's running in memory.

🔁 Difference Between Program and Process:

Feature	Program	Process
State	Passive (just code)	Active (running instance)
Stored In	Disk (secondary storage)	RAM (main memory)
Lifespan	Exists permanently unless deleted	Exists temporarily while executing
Purpose	Represents logic or task	Performs the task using system resources
Multiples	One program can create multiple processes	Each process runs separately with its own memory
Example	`chrome.exe` on disk	Multiple Chrome tabs = multiple running processes

---

☑️ Summary:

- A **program** is like a recipe.
- A **process** is like a chef actively cooking using that recipe.

🧠 Processes Communicating over a Network

A **process** is a **program in execution** running within a host (computer).

💻 Communication Types:

- 🔁 **Within the same host:**
  Processes communicate using **inter-process communication (IPC)** — handled by the operating system (e.g., shared memory, pipes).
- 🌐 **Between different hosts:**
  Processes communicate by **exchanging messages** over the network.

🔗 Client and Server Roles:

- **Client process:**
  The process that **initiates the communication**.
  Example: Your web browser.
- **Server process:**
  The process that **waits to be contacted**.
  Example: A web server that responds with a webpage.

🔁 P2P (Peer-to-Peer) Applications:

- Even in P2P architecture, some processes **act as clients** (requesting data) and others **act as servers** (providing data).
- These roles may **change dynamically** based on need.

☑ Summary:

Processes in different hosts talk through **message exchanges**.
Clients start the conversation, **servers** respond.
In P2P systems, each host can be **both** client and server.

🌐 Process-to-Process Communication Between Remote Hosts

When two processes run on **different hosts** in a network (like over the Internet), they must communicate by **exchanging messages**. This is known as **remote process communication**.

How does it work?

To enable this communication:

- A **client process** initiates the message.
- A **server process** waits to be contacted.

But they **can't talk directly** like two apps on the same computer — they need a communication endpoint called a **socket**.

🧱 What is a Socket?

A **socket** is like a **doorway** through which a process sends or receives messages over the network.
It acts as the **interface between the application layer and the transport layer**.

So for process-process communication over the internet:

1. A **socket** is created on both client and server sides.
2. The **client sends a request** to the server's socket (IP + port number).
3. The **server socket accepts** the request and sends back a response.

🔑 Summary:

- Remote processes communicate by **exchanging messages**.
- A **socket** must be created on both client and server sides.
- This is how a **client process and server process communicate over a network**.

Two processes communicate with each other over a network using sockets.

🧠 Explanation:

A **socket** acts as a **communication endpoint** between two processes. It provides the interface through which data is sent and received between:

- A **client process** (initiates connection)
- A **server process** (waits for incoming connection)

Together, the client and server **establish a connection** using their sockets, and **exchange messages** over that connection.

🔁 Simple Summary:

Socket = bridge between two communicating processes over a network.

A **socket** is located **between the application layer and the transport layer** in the network protocol stack.

🧠 Explanation:

- The **application layer** (e.g., web browser, email app) wants to send or receive data.
- The **socket** acts as the **interface** through which the application **accesses network services** provided by the **transport layer** (like TCP or UDP).

Socket = bridge between the app and the network.
It allows **processes to send/receive messages** using transport protocols like TCP or UDP.

🌐 Web Browser Connections & Sockets

When we use a **web browser**, multiple connections are established to retrieve different parts of a webpage (like images, text, CSS, etc.).

- For **each connection**, a **separate application thread** is used.
- Also, **each connection has its own unique socket**.

Each **socket** is identified by a combination of:

- **IP address**
- **Port number**

This combination helps to **uniquely identify a specific application process** running on a host machine, **even globally**.

🔑 Key Point:

Sockets (IP + port) are used to uniquely identify each connection and direct data to the correct application.

📘 Definition of Socket (from the book):

"A **socket** is the interface between the **application layer** and the **transport layer** within a host. It is also referred to as the application's communication endpoint.

It connects the app to the network and is essential for **process-to-process communication** over the Internet.

For **every network communication**, **two sockets** are involved:

- One is the **sender's socket**, used by the application process that initiates the communication.
- The other is the **receiver's socket**, used by the application process that receives the data.

Together, these two sockets **establish a connection** and enable **process-to-process communication** over the network.

🧠 Key Point:

Just like a phone call needs two phones, a **network connection** needs two sockets — one at each end.

Identifying Sockets in Communication

To identify a **specific socket** (i.e., the communication endpoint) during a network communication, **two types of addresses** are required:

1. **IP Address**:
    - Identifies the **host machine** on the network.
    - Examples:
        - **IPv4** is 32-bit
        - **IPv6** is 128-bit
2. **Port Number**:
    - Identifies the **specific process (application)** running on that host.
    - Each process that wants to communicate is associated with a **unique port number**.

Together, the **IP address and port number form a tuple**, which is referred to as a **socket**.

🧠 Example:

Socket = (IP address, Port number)
This uniquely identifies where the message should be delivered, not just to a device but to the correct application on that device.

What Identifies a Socket?

A **socket** is identified by two things:

- The **IP address** of the host machine
- The **port number** of the process associated with the communication

Together, this **(IP address, port number)** pair **uniquely identifies the socket** on a particular machine.

🧠 **In short**:

A **socket** = IP address + Port number → this combination ensures that data reaches the correct process on the correct host.

Suppose you're visiting a website using a browser (client), and the website is hosted on a server.

- The **server's IP address** is: `142.250.190.78`
- The **server process** (like a web server) is running on **port 80** (HTTP)

🔗 Then the **server socket** is:

```
(142.250.190.78, 80)
```

This means the client's data will be sent to the **web server process** (port 80) on the machine with IP `142.250.190.78`.

📡 What Does an Application-Layer Protocol Define?

An **application-layer protocol** specifies how networked applications communicate. It defines:

1. **Types of Messages Exchanged**
   – Such as **requests**, **responses**, **commands**, etc.

   – What kind of messages are sent (like **request** and **response**)
   – Example: HTTP uses GET and POST messages.

2. **Message Syntax**
   – The **structure** or **format** of messages
   – What fields exist in the message and how they are **separated (delimited)**
3. **Message Semantics**
   – The **meaning** of each field
   – What information the fields carry and how it should be interpreted

4. **Rules of Communication**
   – When and how processes should **send**, **receive**, or **respond** to messages

---

🌐 Types of Protocols

Open Protocols

- Defined publicly in **RFCs (Request for Comments)**
- Anyone can access and implement them
- Promotes **interoperability** between different systems
- Examples: **HTTP**, **SMTP**, **DNS**

Proprietary Protocols

- Owned and controlled by private companies
- Not openly published
- Only the company's software can fully understand or implement it
- Examples: **Skype**, **Zoom**

What Transport Service Does an Application Need?

Applications have different requirements from the transport layer depending on their purpose:

1. **Data Integrity (Reliability):**
   - Some applications like **file transfer** and **web transactions** need **100% reliable data transfer**.
   - They require every bit of data to arrive **correctly and in order** without loss.
2. **Loss Tolerance:**
   - Other applications such as **audio streaming** or **video conferencing** can tolerate some data loss.
   - For these, occasional loss of data packets does not severely affect the user experience.
3. **Timing (Delay Sensitivity):**
   - Applications like **Internet telephony (VoIP)** and **interactive games** need **low delay** to function effectively.
   - High latency or delays reduce the quality and responsiveness, making the application unusable.

- **Throughput:**

  - Some applications, like **multimedia streaming**, require a **minimum amount of throughput** (data rate) to work effectively without interruptions.

- Other applications, called **elastic apps** (e.g., file downloads, emails), can adapt and make use of **whatever throughput** is available, adjusting their speed accordingly.

- **Security:**
  - Many applications need **encryption** and other security features to protect data from unauthorized access during transmission.
  - Transport layer protocols or additional layers provide mechanisms to ensure **confidentiality, integrity, and authentication**.

Transport Service Requirements: Common Applications

Application	Data Loss Tolerance	Throughput Requirement	Time Sensitivity
File transfer / download	No loss	Elastic	No
E-mail	No loss	Elastic	No
Web documents	No loss	Elastic	No
Real-time audio/video	Loss-tolerant	Audio: 5 Kbps – 1 Mbps	Yes, 10's of milliseconds
Streaming audio/video	Loss-tolerant	Video: 10 Kbps – 5 Mbps	Yes, few seconds
Interactive games	Loss-tolerant	Kbps+	Yes, 10's of milliseconds
Text messaging	No loss	Elastic	Yes and no (varies)

🌐 Internet Transport Protocol Services

(*Based on Kurose & Ross – Computer Networking: A Top-Down Approach*)

🧩 TCP (Transmission Control Protocol)

TCP is a **connection-oriented**, **reliable transport layer protocol** that provides several essential services:

☑ Key Features of TCP:

1. **Reliable Data Transfer**
   - TCP guarantees that data sent by the sender will be delivered **accurately and in order** to the receiving process.

   - It uses mechanisms such as **ACKs (Acknowledgments)**, **retransmissions**, and **sequence numbers** to ensure reliability.
2. **Flow Control**
   - Prevents the **sender from overwhelming the receiver**.
   - TCP uses a **sliding window protocol** and **receiver's advertised window** to dynamically adjust the rate of data transmission.
3. **Congestion Control**
   - Protects the network from being overloaded with traffic.
   - TCP monitors **network conditions** and **throttles the sender's rate** during congestion (e.g., using algorithms like AIMD – Additive Increase Multiplicative Decrease).
4. **Connection-Oriented Communication**
   - A **three-way handshake** is required before data transmission can begin.
   - This setup phase creates a **logical connection** between the client and server processes.

✗ *What TCP Does* Not *Provide:*

- **Timing Guarantees:** No assurance on when data will arrive (not suitable for real-time).
- **Minimum Throughput Guarantees:** Cannot reserve or guarantee bandwidth.
- **Security:** No built-in encryption or authentication (can be added using TLS/SSL on top of TCP).

🧩 UDP (User Datagram Protocol)

UDP is a **connectionless**, **lightweight** transport protocol. It is often referred to as a **"best-effort" delivery service**.

☑ *Key Characteristics of UDP:*

- **No connection setup:** Sender can start sending data immediately.
- **Unreliable transfer:** No acknowledgments, no retransmissions.
- **No flow or congestion control:** Sender does not adjust sending rate based on receiver or network state.
- **No ordering:** Packets may arrive out of order or not at all.

✗ *What UDP Does* Not *Provide:*

- Reliability
- Flow control
- Congestion control
- Connection setup
- Timing or delay guarantees
- Security or encryption

❓ Why Bother Using UDP?

Despite its limitations, UDP is **essential** for several real-world applications due to:

1. **Low Overhead and Simplicity:**
   - No connection setup means less delay and less protocol complexity.
   - Ideal for applications where quick startup and fast delivery are more important than perfect reliability.
2. **Speed and Efficiency:**
   - Useful in **real-time applications** (e.g., **VoIP, video conferencing, online gaming**) where dropping a few packets is acceptable, but delays are not.
3. **Allows Custom Reliability:**
   - Applications like **DNS** or **live media streaming** may implement their own reliability mechanisms only when necessary, giving developers more **control and flexibility**.
4. **Small Packet Size:**
   - UDP headers are only 8 bytes (vs. 20 bytes for TCP), reducing overhead.

📌 **In summary**:
UDP is chosen when **speed, low latency, and minimal protocol overhead** are more important than **reliability and order**.

📚 Real-World Application Examples (from Kurose & Ross):

Application	Protocol Used	Reason
Web Browsing (HTTP)	TCP	Requires reliable, ordered delivery of content
Email (SMTP, IMAP)	TCP	Delivery assurance and complete message reception needed
Video Streaming (YouTube, Netflix)	Mostly TCP (with buffering)	Ensures complete and correct delivery of video content
Online Games	UDP	Low latency preferred; occasional loss tolerated
Voice Calls (VoIP)	UDP	Real-time performance more critical than perfect delivery
DNS Queries	UDP	Very fast response time needed; retry handled at application level

---

In short,

Internet Transport Protocols: TCP vs UDP

TCP (Transmission Control Protocol)

- **Provides:**
  - ☑ **Reliable data transfer** (ensures all data arrives correctly and in order)
  - ☑ **Flow control** (prevents sender from overwhelming the receiver)
  - ☑ **Congestion control** (slows sender if the network is congested)
  - ☑ **Connection-oriented** (requires a setup phase before communication)
- **Does NOT provide:**
  - ❌ Timing guarantees
  - ❌ Minimum throughput guarantees
  - ❌ Security (encryption/authentication)

UDP (User Datagram Protocol)

- **Provides:**
  - ☑ **Unreliable** data transfer (no guarantee of delivery, ordering, or duplicates)
- **Does NOT provide:**
  - ❌ Reliability
  - ❌ Flow control
  - ❌ Congestion control
  - ❌ Timing
  - ❌ Throughput guarantees
  - ❌ Security
  - ❌ Connection setup

❓ Why is UDP used at all?

Even though UDP is unreliable, it offers **key advantages**:

- ⚡ **Low latency**: No connection setup = faster communication
- 🧠 **Simplicity**: Lightweight and easy to implement
- 📡 **Useful for real-time applications**: Streaming, gaming, VoIP — where **speed is more important than reliability**
- 🔄 **App-level control**: Applications can implement their own reliability if needed

📝 Summary: **UDP is chosen when speed, simplicity, and low overhead matter more than guaranteed delivery.**

🔒 Securing TCP Connections

1. The Problem with Vanilla TCP & UDP

- **Vanilla sockets** (regular TCP/UDP connections) do **not provide encryption** by default.
- Sensitive information like **passwords**, personal data, or credentials are sent in **plaintext**.
- These **cleartext messages** travel across the Internet and can be **intercepted** by attackers using packet sniffers or man-in-the-middle attacks.
- ⚠️ **Major security concern**: Anyone with access to the communication channel can **read or manipulate** the data.

---

2. Solution: Transport Layer Security (TLS)

☑ *What TLS Provides:*

- **Encryption:** Converts plaintext data into ciphertext before it leaves the device.
- **Data Integrity:** Ensures data hasn't been tampered with during transit.
- **Authentication:** Verifies the identity of the communication parties (e.g., using certificates).

☑ So, even if data is intercepted, it appears meaningless without decryption keys.

---

3. TLS Architecture and Operation

- **TLS is implemented at the Application Layer** (not directly at the transport layer).
- Applications access **TLS through libraries** like:
  - **SSL (Secure Socket Layer)** – older version of TLS.
  - **OpenSSL**, **Java Secure Socket Extension (JSSE)**, etc.
- These libraries work **on top of TCP**, providing a secure interface to the application

🔁 Quick Review of the Web

- A **web page** is made up of **multiple objects**.
- Each object can be:
  - An **HTML file**
  - A **JPEG image**, **GIF**
  - A **JavaScript** or **Java applet**

    - An **audio** or **video file**, etc.
- A typical web page includes:
    - A **base HTML file**
    - Plus **several referenced objects** (like images, scripts, stylesheets)

📍 Addressing Web Objects

Each object is accessible via a **URL (Uniform Resource Locator)**:
Example:
`http://www.someschool.edu/someDept/pic.gif`

- **www.someschool.edu → Host name**
- **/someDept/pic.gif → Path name**

🧩 What is HTTP?

- **HTTP (Hypertext Transfer Protocol)** is the **Web's application-layer protocol**.
- It defines how **clients and servers** communicate to exchange web resources.

👩‍💻 👨‍💻 HTTP Follows the Client-Server Model

Client:

- Typically a **web browser** (e.g., Chrome, Firefox)
- **Requests** web objects from the server using HTTP
- **Receives and displays** the content

Server:

- A **web server** (e.g., Apache, Nginx)
- **Listens** for client HTTP requests
- **Sends back** the requested web objects (e.g., HTML files, images)

🔗 Key Characteristics of HTTP

- **Uses TCP** as the underlying transport protocol (usually port **80**, or **443** for HTTPS).
- **Stateless Protocol**:
  - The server does not retain information about past client requests.
- **Text-based**:
  - All HTTP commands and responses are in **human-readable ASCII text**.

🌐 HTTP Overview – *According to Kurose and Ross, Chapter 2*

🔗 HTTP and TCP:

- **HTTP (Hypertext Transfer Protocol)** is the **Web's application-layer protocol**.
- HTTP uses **TCP as the transport-layer protocol**.
  - A **client initiates** a **TCP connection** to the server at **port 80**.
  - Once the connection is established, **HTTP messages** (requests/responses) are exchanged.
  - The **TCP connection is then closed** (unless using persistent connections in HTTP/1.1).

HTTP is Stateless:

- **Stateless** means that the **server does not retain information** about past client requests.
- Each HTTP request is **independent of the previous one**.
- While this makes the protocol **simpler and faster**, it can be a limitation for applications that need to maintain state (like shopping carts or login sessions).

Client-Server Model:

- **Client**: typically a web browser that **sends HTTP requests** and **displays content**.
- **Server**: a web server that **receives requests** and **responds with web content** (like HTML files, images, videos).

⚠️ Why Maintaining "State" Is Hard

- **Stateful protocols** need to **store and manage context/history** between messages.
- This makes the protocol **more complex**, because:
  - The **server must remember** previous interactions with each client.
  - If either the **client or server crashes**, the stored state may become **inconsistent**.
  - Additional logic is needed to **recover or synchronize** state.

🔗 HTTP Connections: Two Types

HTTP defines how web objects are transferred over TCP. There are two types of connections used for this:

---

1. Non-Persistent HTTP

- For **each requested object**, a **new TCP connection** is opened.
- Once the object is sent, the **TCP connection is closed** immediately.
- If a web page includes multiple objects (like images, CSS, JS), the browser must:
  - **Initiate a new TCP connection** for **each object**.
  - This means **multiple TCP handshakes**, increasing **latency**.

Steps:

1. Client opens TCP connection.
2. Sends an HTTP request for **one object**.
3. Server sends response.
4. TCP connection is **closed**.
5. Repeat for each additional object.

❌ *Drawbacks:*

- **Slow** due to repeated TCP setups and teardowns.
- **Inefficient** in terms of overhead.

---

2. Persistent HTTP

- A **single TCP connection** is **opened** and used to transfer **multiple objects** between client and server.
- After sending the base HTML file, the same connection is used to request and receive **all additional resources** (e.g., images, scripts).
- The connection is **closed** only after all the required objects are transferred, or after a timeout.

☑️ *Advantages:*

- **Reduced latency** (fewer handshakes).
- **Improved performance**.
- **Lower overhead** on both client and server.

💡 *Note:*

- HTTP/1.1 uses **persistent connections by default**.
- Non-persistent behavior must be explicitly requested with `Connection: close`.

⏱️ RTT (Round Trip Time)

RTT is the **time it takes for a small packet to go** from the **client to the server** and back to the **client**. It's like saying "hello" and waiting for the "hello back."

🌐 HTTP Response Time (Non-Persistent HTTP)

To receive **one object** (like an image or HTML file), the time includes:

1. **1 RTT** to **set up the TCP connection**
2. **1 RTT** for the **HTTP request + first few bytes of the response**
3. **Transmission time** to send the entire file (depends on file size and bandwidth)

🧮 So, Total Response Time = 2 RTT + File Transmission Time

```
Total time = 1 RTT (TCP handshake)
           + 1 RTT (request & first response)
           + transmission time
           = 2 RTT + file transmission time
```

📝 **Note:**
This is for **non-persistent HTTP**, where **each object** requires a **separate TCP connection**.
So, if a page has 10 objects, it'll need **10 × (2 RTT + transmission time)**.

🌐 Non-Persistent HTTP (Problems)

- Requires **2 RTTs** for **every single object**.
- **Opens and closes** a new TCP connection for each object.
- This creates **overhead** for the operating system.
- Browsers try to reduce delay by opening **multiple parallel TCP connections**, but this increases complexity.

🔁 Persistent HTTP (HTTP 1.1 Solution)

- Only **1 TCP connection** is opened and **kept alive**.
- Server **doesn't close** the connection after sending a response.
- **Multiple HTTP requests/responses** can be sent over the **same connection**.
- As soon as the browser encounters a referenced object (like images, CSS), it can send a request **without waiting**.
- This reduces response time significantly — **possibly just 1 RTT** for all objects!

☑️ Benefits of Persistent HTTP

- **Faster page loading**
- **Less overhead**
- **Fewer TCP handshakes**
- **Efficient use of network resources**

🌐 HTTP Request Message

In the **HTTP protocol**, when a browser wants to fetch something (like a webpage or image) from a server, it sends an **HTTP request message**.

Types of HTTP Messages:

- **Request**: Sent by the client (browser) to the server.
- **Response**: Sent by the server back to the client.

Structure of an HTTP Request Message

The request message is **text-based** and human-readable (ASCII format), and it has the following parts:

1. **Request Line**
   Example:

   ```
   GET /index.html HTTP/1.1
   ```

   - `GET` → The method (can also be `POST`, `HEAD`, etc.)
   - `/index.html` → The resource being requested
   - `HTTP/1.1` → The version of HTTP used
2. **Header Lines**
   These give **extra information** about the request:

   ```
   Host: www-net.cs.umass.edu
   User-Agent: Mozilla/5.0 ...
   Accept: text/html,application/xhtml+xml
   Accept-Language: en-us,en;q=0.5
   Accept-Encoding: gzip,deflate
   Connection: keep-alive
   ```

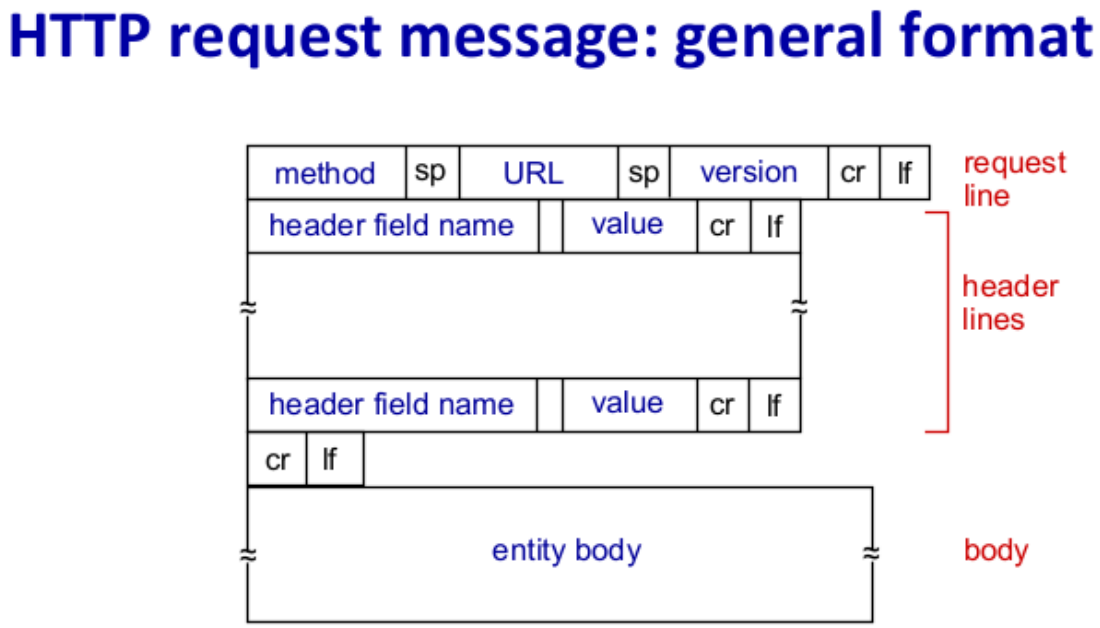
   - `Host`: Specifies the website domain.
   - `User-Agent`: Info about the browser and OS.
   - `Accept`: What types of content the browser can accept.
   - `Connection: keep-alive`: Tells the server to keep the connection open.
3. **Blank Line (`\r\n`)**
   A carriage return and line feed **end the header section** and separate it from the body (if there is one, like in POST requests).

Example Summary:

```
GET /index.html HTTP/1.1
Host: www-net.cs.umass.edu
User-Agent: Mozilla/5.0
Accept: text/html
Connection: keep-alive
```

☑ Full Example:

```
POST /login HTTP/1.1\r\n
Host: www.example.com\r\n
Content-Type: application/x-www-form-urlencoded\r\n
Content-Length: 29\r\n
\r\n
username=irfan&password=1234
```



🧠 **Note**: In most GET requests, there is no body—just the request line and headers.

This **carriage return (`\r`) and line feed (`\n`)** combo:

- 📍 **Marks the end of the header section**
- 📍 **Separates the header from the body** (if the message has a body — usually in `POST`, `PUT` requests)

🔍 Other HTTP Request Methods (Based on Kurose & Ross)

In addition to the common **GET** request used to fetch web pages, the HTTP protocol supports several other types of request methods. These are useful for different forms of communication between the client (browser) and the server. Below are detailed descriptions of the important ones:

🔹 1. POST Method (HTTP 1.0)

- The **POST method** is used when a client (usually a browser) wants to send **data to the server** that is not suitable for a URL, such as **form data**.
- Instead of appending data to the URL (as in GET), POST sends the data in the **body** (or "entity body") of the HTTP request.
- This method is **secure for sensitive data** (e.g., passwords), as it doesn't expose them in the URL.
- **Common use case:** Submitting forms, logging in, or posting comments.

Example:

```
http
CopyEdit
POST /submit-form HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 35

username=siam&password=123456
```

🔹 2. GET Method (HTTP 1.0) — with Query Parameters

- The **GET method** is used to **request data** from a server.
- When sending small amounts of data (e.g., search terms), it can be **included directly in the URL** using a `?` followed by key-value pairs.
- This method is **simple** but not recommended for sensitive information since the data is **visible in the URL** and may be logged or cached.
- **Common use case:** Retrieving pages based on user input like search queries.

Example:

```
GET /search?animal=monkey&color=brown HTTP/1.1
Host: www.example.com
```

🔹 3. HEAD Method (HTTP 1.0)

- The **HEAD method** is similar to a GET request, but the server **only sends the header lines** back — **not the body** (content) of the response.
- This is useful when a client wants to **check the metadata** of a resource (such as file size or last modification date) without downloading the entire content.
- It helps in **saving bandwidth** and checking resource availability.

Use case: Check if a webpage has changed before downloading it.

Example:

```
HEAD /report.pdf HTTP/1.1
Host: www.example.com
```

🔹 4. PUT Method (HTTP 1.1)

- The **PUT method** allows the client to **upload a new file** or **update an existing resource** on the server.
- The contents of the new or updated file are included in the **entity body** of the request.
- If a file already exists at the specified URL, it will be **replaced** with the new one.

Common use case: Used by developers or applications for file storage or version control.

Example:

```
PUT /uploads/assignment.docx HTTP/1.1
Host: www.example.com
Content-Type: application/msword
Content-Length: 10240

(binary content of the file goes here)
```

🌐 HTTP Response Message (Based on Kurose & Ross)

When a client (like a web browser) sends an HTTP request to a server, the server replies with an **HTTP response message**. This message contains three main parts:

🧩 1. Status Line

The first line in the HTTP response is the **status line**. It tells the client the result of the request and consists of:

- **HTTP version** (e.g., HTTP/1.1)
- **Status code** (e.g., 200, 404, 503)
- **Status phrase** (e.g., OK, Not Found, Service Unavailable)

Example:

```
HTTP/1.1 200 OK
```

This means the request was successful and the server is sending the requested resource.

🧩 2. Header Lines

Header lines provide additional information about the response, such as:

- The date and time of response
- Server software details
- When the resource was last modified
- Content type and length

Example:

```
Date: Tue, 08 Sep 2020 00:53:20 GMT
Server: Apache/2.4.6 (CentOS)
Last-Modified: Tue, 01 Mar 2016 18:57:50 GMT
Content-Length: 2651
Content-Type: text/html; charset=UTF-8
```

Each header line ends with `\r\n` (carriage return and line feed).

🧩 3. Blank Line

A **blank line (\r\n)** follows the header section. It separates the headers from the actual content (body) of the response.

🧩 4. Entity Body (Data)

The entity body contains the **actual data** the client requested, such as an HTML page, image, video, or any other file.

Example (HTML file contents):

```
<html>
  <head><title>Welcome</title></head>
  <body><h1>Hello, World!</h1></body>
</html>
```

🧠 Summary Table:

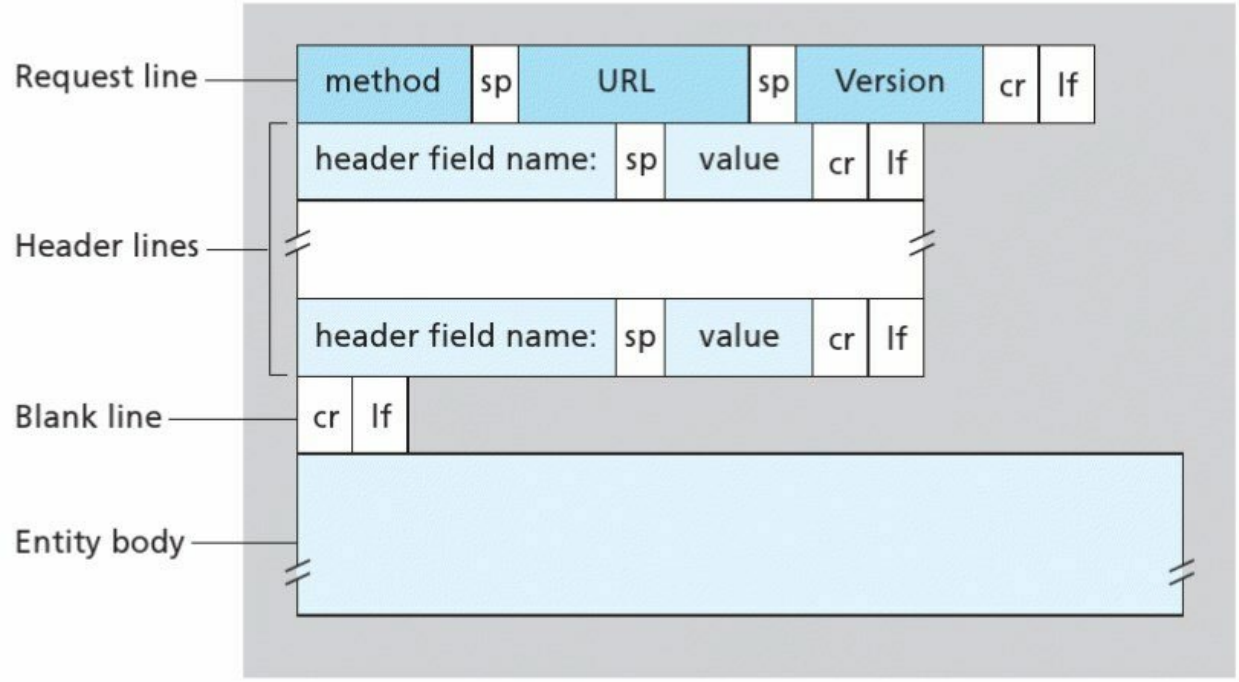
Section	Description
Status Line	Shows protocol version, status code, and phrase
Header Lines	Extra info like date, server, content type, content length
Blank Line	Marks the end of headers
Entity Body	Actual content (HTML, image, etc.) sent in response

---

📌 Example Full Response (Simplified):

```
HTTP/1.1 200 OK
Date: Tue, 08 Sep 2020 00:53:20 GMT
Server: Apache
Content-Length: 100
Content-Type: text/html

<html>
  <body>Welcome to our website</body>
</html>
```



🌐 HTTP Response Status Codes

HTTP status codes are **3-digit numbers** sent in the **first line** of the response message from the server to the client. They indicate the **result of the client's request**.

These codes are grouped into different categories:

☑ 1xx – Informational

Temporary responses. Rarely used in practice.

☑ 2xx – Success

- **200 OK**
  → The request was successful. The requested object is included in the response.

🔁 3xx – Redirection

- **301 Moved Permanently**
  → The requested resource has been permanently moved to a new location.
  → The new URL is provided in the `Location:` header of the response.

🚫 4xx – Client Errors

- **400 Bad Request**
  → The request was malformed or had syntax errors. The server couldn't understand it.
- **404 Not Found**
  → The server can't find the requested resource. It may not exist or the URL is incorrect.

❌ 5xx – Server Errors

- **500 Internal Server Error**
  → The server had a problem while trying to fulfill the request.
- **503 Service Unavailable**
  → The server is temporarily overloaded or down for maintenance.

🧠 Example (First Line of Response):

```
HTTP/1.1 200 OK
```

or

```
HTTP/1.1 404 Not Found
```

📝 Summary Table

Code	Meaning	Explanation
200	OK	Request succeeded
301	Moved Permanently	Resource moved to a new URL
400	Bad Request	Syntax or format error in client's request
404	Not Found	Resource does not exist on server
500	Internal Server Error	Server had a problem processing the request
503	Service Unavailable	Server is busy or temporarily unavailable

🍪 Maintaining User/Server State: Cookies

📌 Problem: HTTP is stateless

- Each HTTP request is **independent**.
- The server does **not remember** any previous interaction with the same client.
- So, multi-step processes like login, cart handling, etc., need a way to **remember user activity**.

☑ Solution: Cookies

Cookies are small pieces of data that allow **state information** to be maintained **across multiple HTTP requests**.

🔄 How Cookies Work

1. **First Visit**
   → Client sends a normal HTTP request to server.
   → Server responds with a **Set-Cookie** header:

   ```
   Set-Cookie: ID=12345
   ```

2. **Subsequent Requests**
   → Client includes the cookie with every request:

   ```
   Cookie: ID=12345
   ```

   → Server uses this ID to retrieve session data (e.g., login status, preferences, cart).

🏷️ What's in a Cookie?

- **Name-Value Pair** (e.g., ID=12345)
- **Expiration Date** (when it should be deleted)
- **Domain and Path** (where it is valid)
- **Security Flags** (e.g., HTTPS-only)

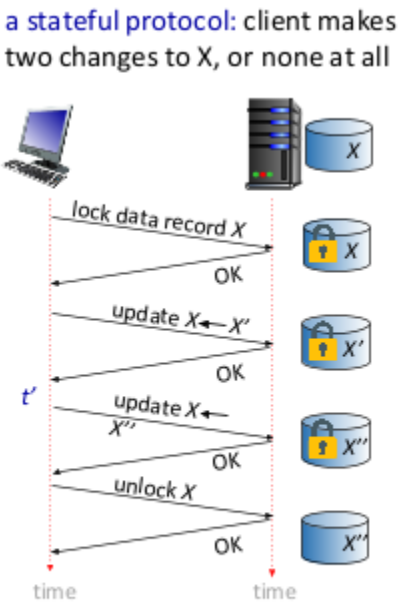
💼 Use Cases of Cookies

- **User authentication** (login sessions)
- **Shopping cart tracking**
- **User preferences**
- **Analytics and tracking**

💥 Issue Without Cookies (Statelessness)

If there's no cookie:

- Server doesn't know who you are.
- Each request feels like it's coming from a **new client**.
- No memory of login, no cart, no personalization.



Q: what happens if network connection or client crashes at *t'* ?

🧩 Stateful Protocol Example: What Happens at `t'`?

This diagram shows a **stateful interaction** between a **client** and a **server**, where the client:

1. Locks a data record **X**
2. Makes two updates:
   - First to **X′**
   - Then to **X″**
3. Unlocks the record

This kind of multi-step exchange **depends on remembering past actions**, which means the **server must keep "state"** of what's going on.

❗ Problem at `t'` — What if Client Crashes?

At time `t'`, the client is **in between** update operations. If it:

- **Crashes**, or
- **Loses connection**

Then the **record X remains locked** with partial updates!

🔍 Why is this a Problem?

- The system is now **inconsistent**:
  The server knows the record is locked, but **doesn't know** whether both updates were completed.
- Other clients can't access **X**, since it's locked.
- This state must now be **manually recovered or reset**, which adds complexity.

☑ Key Takeaway

- **Stateful protocols** like this need to **remember** past steps.
- They can get stuck or confused if the communication **breaks mid-way**.
- That's why HTTP was designed to be **stateless** — easier to manage without this kind of risk.

🍪 Maintaining User/Server State: Cookies

The **HTTP protocol is stateless**, meaning it does not keep track of previous interactions. To overcome this limitation and maintain user-specific information (like login sessions or shopping cart content), **cookies** are used.

How Cookies Work (4 Components):

1. **HTTP Response Cookie Header**
   - When a user visits a website, the server responds with an HTTP response that includes a `Set-Cookie:` header.
   - Example:

     ```
     Set-Cookie: userID=12345
     ```

2. **HTTP Request Cookie Header**
   - On future visits, the browser automatically includes the cookie in the `Cookie:` header of HTTP requests to the same server.
   - Example:

     ```
     Cookie: userID=12345
     ```

3. **Cookie File on Client**
   - The user's browser stores the cookie on the user's device (hard drive or memory).
   - Cookies can have expiration dates and are managed by the browser.
4. **Back-End Database**

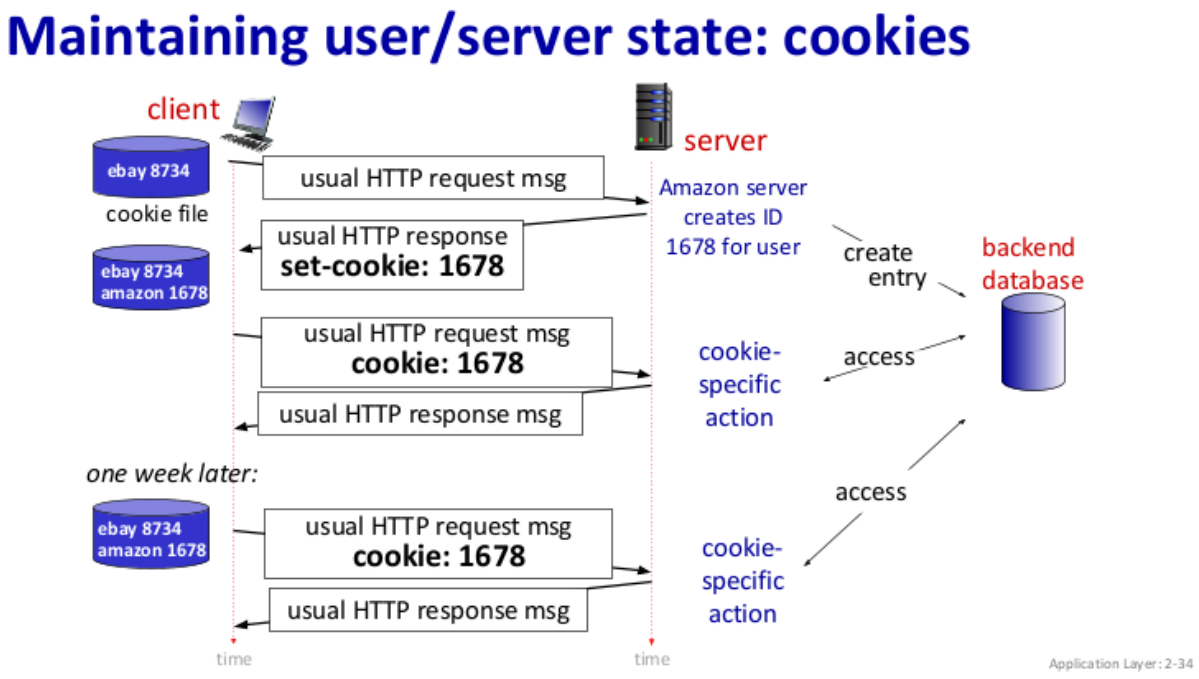
- The website stores cookie-related data (e.g., user info, preferences) in a server-side database, linked to the unique cookie ID.

🧩 Example (from the book):

- Susan uses her browser to visit an e-commerce site for the first time.
- The server:
  - Generates a unique cookie ID for Susan (e.g., `ID=AB1234`)
  - Stores this ID and related session info in its backend database.
- Susan's browser:
  - Stores the cookie and includes it in all future requests to the site.
- Result:
  - The server can **recognize** Susan on each visit and provide a personalized experience (e.g., saved cart, login status).

☑ Benefits:

- Enables **session management**, **user tracking**, and **personalization**.
- Essential for **login systems**, **shopping carts**, **analytics**, etc.



🍪 HTTP Cookies: Comments and Uses

☑ What Cookies Are Used For:

Cookies help maintain **state** between different HTTP requests in a **stateless** protocol like HTTP. They are used for:

- **Authorization** – remembering login sessions.
- **Shopping Carts** – keeping track of items even if the user navigates away.
- **Recommendations** – storing preferences or browsing behavior.
- **User Session State** – like in webmail (Gmail, Yahoo Mail), to manage continuous user sessions.

⚠ Cookies and Privacy:

Cookies can also raise **privacy concerns**:

- **Tracking Your Behavior**: Sites can learn about your actions and preferences on their platform.
- **Third-party (Tracking) Cookies**:
  - Placed by advertisers or analytics tools.
  - Allow tracking the same user across **multiple websites**.
  - Example: You search for shoes on one site and see shoe ads on another — enabled by third-party cookies.

🔁 Challenge: How to Maintain State in HTTP?

Since HTTP itself is **stateless**, cookies solve this in two main ways:

1. **At protocol endpoints** (server/client):
   - The server stores session-related info (like login status) mapped to a cookie ID.
2. **Inside messages**:
   - Cookie data is **sent with every HTTP message** (request and response), allowing the server to maintain continuity.

🍪 What is a Cookie (in Computer Networking / Web)?

Definition:
A **cookie** is a small piece of data stored on the user's browser by a website. It helps websites remember information about the user across sessions.

🌐 Web Caches (Proxy Server Caching)

📌 What is a Web Cache?

A **Web cache** (also called a **proxy server**) is a network entity that stores copies of **frequently accessed web content** (like HTML files, images, videos) temporarily to reduce access delay and save bandwidth.

🧠 How It Works:

1. 🔧 **Browser is configured** to send all HTTP requests to the cache (instead of the original server).
2. 📁 **If the requested object is in the cache**:
   - Cache immediately returns it to the browser (fast access!).
3. 🌍 **If the object is NOT in the cache**:
   - Cache requests it from the **origin server**.
   - Then stores a **copy** of the object.
   - Sends it back to the client.

🎯 Goal of Web Caching:

- Reduce **response time** for the client.
- Minimize **traffic** to the origin server.
- Improve **efficiency** of content delivery.

🌐 Web Caches (Proxy Servers)

A **Web cache** is a **network server** that stores **frequently requested web content** to make future requests faster and reduce traffic on the internet.

🔄 How It Works:

- The **web cache** acts as both:
  - A **server** to the **client** (provides the requested object).
  - A **client** to the **origin server** (fetches the object if not cached).
- **If the object is cached**, it is delivered quickly to the user.
- **If not**, it is fetched from the original server, stored, and then sent.

💬 Server Control on Caching:

Web servers can **control how caching works** using response headers:

- `Cache-Control: max-age=<seconds>` → cache can keep the object for that many seconds.
- `Cache-Control: no-cache` → cache must revalidate before using the cached copy.

❓ Why Use Web Caching?

☑ **Reduce client response time** (cache is closer to user).
☑ **Reduce network traffic** (especially for organizations).
☑ **Support small content providers** (caches help deliver their content effectively).

Option 2: install a web cache

Scenario:

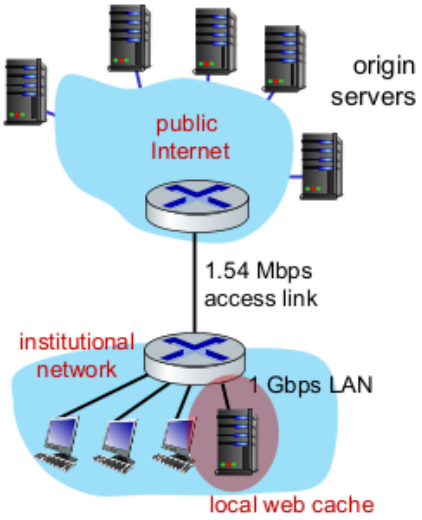
- access link rate: 1.54 Mbps
- RTT from institutional router to server: 2 sec
- web object size: 100K bits
- average request rate from browsers to origin servers: 15/sec
  - avg data rate to browsers: 1.50 Mbps

Cost: web cache (cheap!)

Performance:

- LAN utilization: .?
- access link utilization = ?
- average end-end delay = ?

How to compute link utilization, delay?



🌐 Caching and Performance in Institutional Networks (Application Layer)

🔹 Scenario Setup

- **Network Configuration:**
  - LAN: 1 Gbps
  - Access Link: 1.54 Mbps (to the Internet)
  - RTT to origin server: 2 seconds
  - Web object size: 100 Kbits
  - Request rate: 15 requests/sec
  - Average data rate to browsers: 1.50 Mbps

🔹 Performance Without Optimization

- **Access Link Utilization:**

  Utilization=1.501.54≈0.97\text{Utilization} = \frac{1.50}{1.54} \approx 0.97Utilization=1.541.50≈0.97

  - High access link utilization (97%)
  - Leads to long **queueing delays**
- **LAN Utilization:**

  LAN Utilization=very low≈0.0015\text{LAN Utilization} = \text{very low} \approx 0.0015LAN Utilization=very low≈0.0015

  - LAN not a bottleneck
- **Average End-to-End Delay:**

  Total Delay=Internet delay+Access link delay+LAN delay=2 sec+minutes+μs\text{Total Delay} = \text{Internet delay} + \text{Access link delay} + \text{LAN delay} = 2 \text{ sec} + \text{minutes} + \mu sTotal Delay=Internet delay+Access link delay+LAN delay=2 sec+minutes+μs

  - Very high delay due to congestion at the access link

🔹 Option 1: Upgrade Access Link (Expensive)

- **Upgrade bandwidth to 154 Mbps**
  - Reduces access link utilization significantly:

    Utilization=1.50154≈0.0097\text{Utilization} = \frac{1.50}{154} \approx 0.0097Utilization=1541.50≈0.0097

    - Minimal queueing delay (milliseconds)
- **Resulting Delay:**
  - LAN Delay: ~μs
  - Internet + Access Link Delay: ~ms
- **Downside:** Very expensive upgrade

🔹 Option 2: Use a Web Cache (Cost-Effective)

Concept:

- Install a **local web cache** in the institutional network
- Frequently accessed content served locally, avoiding the Internet

---

☑ Calculating Access Link Utilization & End-to-End Delay With Cache

- **Assume Cache Hit Rate:** 40%
  → 40% served by local cache (low delay)
  → 60% go to origin server (via Internet)

Access Link Usage:

- Only 60% of requests use the Internet:

  Rate to browsers over access link=0.6×1.50 Mbps=0.9 Mbps\text{Rate to browsers over access link} = 0.6 \times 1.50 \text{ Mbps} = 0.9 \text{ Mbps}Rate to browsers over access link=0.6×1.50 Mbps=0.9 Mbps

- **Utilization:**

  Utilization=0.91.54≈0.58\text{Utilization} = \frac{0.9}{1.54} \approx 0.58Utilization=1.540.9≈0.58

  - Lower than 0.97 → implies **low queueing delay**

Average End-to-End Delay:

- Delay from origin = ~2.01 sec (including queueing)
- Delay from cache = ~milliseconds

Average delay=0.6×2.01+0.4×(msec)≈1.2 sec\text{Average delay} = 0.6 \times 2.01 + 0.4 \times \text{(msec)} \approx 1.2 \text{ sec}Average delay=0.6×2.01+0.4×(msec)≈1.2 sec

Conclusion:

- **Much lower delay** than before (1.2 sec vs. 2+ minutes)
- **Cheaper** than upgrading to 154 Mbps

🌐 Conditional GET in HTTP

Goal:
Avoid sending a web object **if the client's cache already has the latest version**.
→ **Saves time and network bandwidth**.

🔁 How It Works

1. **Client** already has a cached copy of a web object.
2. When it makes a request to the **server**, it adds a header:

```
bash
CopyEdit
If-Modified-Since: <date>
```

→ This tells the server:
"I already have a version of this object saved from `<date>`. Has it changed?"

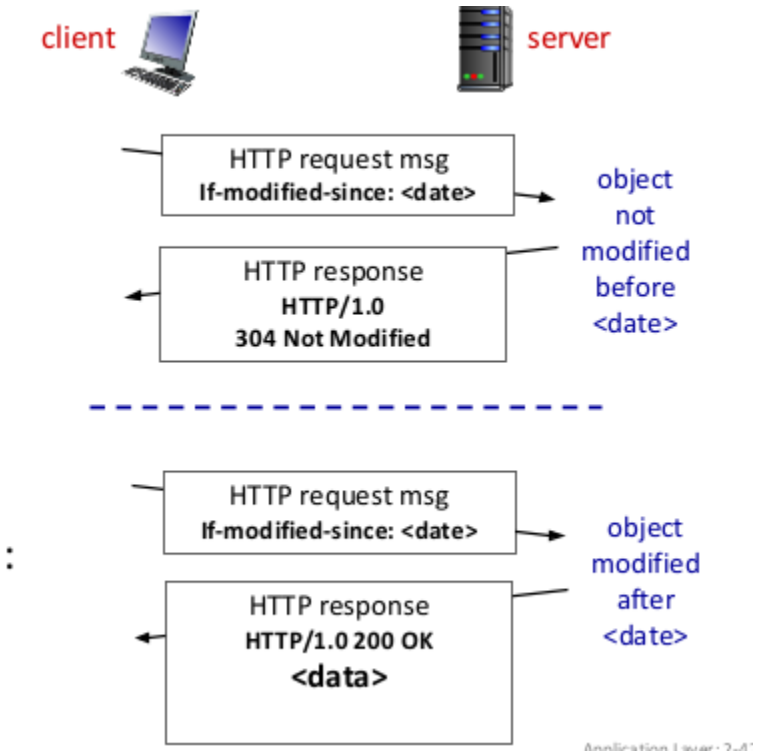
3. **Server checks** if the object has been modified since that date:
   - ☑ **If modified** → server sends the **new object**.
   - ❌ **If not modified** → server replies:

```
mathematica
CopyEdit
HTTP/1.0 304 Not Modified
```

→ No object is sent.

☑ Benefit:

- Reduces unnecessary data transfers.
- **Faster response** for users.
- **Less load** on servers and networks.



📘 HTTP/2 Overview – Reducing Delay in Multi-Object Requests

🔹 Background: HTTP/1.1

- Supports **multiple pipelined GET requests** over a single TCP connection.
- Server responds **in order (First-Come-First-Served, FCFS)** to requests.
- Problem: **Head-of-Line (HOL) blocking**
  - Small objects can get stuck waiting behind large objects.
- **Loss recovery stalls** TCP transmission until lost segments are retransmitted.
- This leads to **higher delays** in loading complex web pages with many objects.

🔹 What's New in HTTP/2? (RFC 7540, 2015)

- Improves flexibility in **sending multiple objects** to client.
- Keeps most HTTP methods, status codes, and headers **unchanged** from HTTP/1.1.
- Key features:
  - **Client-specified priority:** Server sends objects based on priority, not just FCFS.
  - **Server push:** Server can proactively push objects to client without explicit request.
  - **Frames:** Objects divided into smaller units called **frames**.
  - **Frame scheduling:** Frames are scheduled to **mitigate HOL blocking**.
- Goal: **Decrease delay for web pages with many objects**, improving performance.

OR,

📗 HTTP/2 Explained: Faster Web with Smarter Object Delivery

🔸 Why was HTTP/2 needed?

Web pages often consist of many small objects like images, scripts, stylesheets, etc. When a browser requests these objects, the speed of loading the page depends on how efficiently the server sends these objects.

🔸 HTTP/1.1: What happens?

- When you open a web page, the browser sends multiple **GET requests** for different objects (like images).

- HTTP/1.1 allows these GET requests to be **pipelined** over a single TCP connection, meaning the client can send multiple requests before receiving responses.
- But the server responds **in the order it received requests** (First-Come-First-Served, FCFS).

Problem:
If a **large object** is requested first, smaller objects behind it in the queue must **wait their turn**. This is called **Head-of-Line (HOL) blocking** — it blocks other objects from being sent, causing delays in page load.

Also, if some TCP segments are lost during transmission, the server must **retransmit** lost data before continuing, which stalls the entire transfer.

🔸 HTTP/2: What's different?

HTTP/2 was designed to **fix these problems** and speed up web page loading.

1. **Multiplexing with Frames:**
   - Instead of sending whole objects one-by-one, HTTP/2 **splits objects into smaller chunks called frames**.
   - Frames from multiple objects are **interleaved** (mixed together) on the same TCP connection.
   - This means the server can send parts of different objects **at the same time** without waiting for one object to finish.
2. **Client-Specified Priorities:**
   - The browser tells the server which objects are **more important** (higher priority).
   - The server sends higher priority objects first, not just in the order requests arrived.
3. **Server Push:**
   - The server can **send objects the client did not explicitly request yet** if it knows the client will need them soon (like CSS files or images referenced by HTML).
   - This avoids round-trip delays where the client has to ask for them separately.
4. **Reducing HOL Blocking:**
   - Because frames from different objects are interleaved, **small objects don't get stuck waiting behind large ones**.
   - Even if some frames are lost, retransmission affects only the lost frames, not the entire TCP stream for all objects.

🔸 How does HTTP/2 improve web performance?

- Faster loading of pages with many objects.
- More efficient use of the single TCP connection.
- Better handling of packet loss.

- Reduced delay from waiting on large objects.

Head-of-Line (HOL) Blocking and HTTP Versions (According to Kurose & Ross)

HTTP/1.1 and HOL Blocking

- HTTP/1.1 allows multiple GET requests over a single TCP connection, using **pipelining**.
- However, the server sends responses **in the order the requests arrive** (First-Come-First-Served).
- If a large object (e.g., video file) is requested first, smaller objects behind it are **blocked and must wait**.
- This is called **Head-of-Line (HOL) blocking** at the application layer.

HTTP/2: Dividing Objects into Frames and Interleaving Transmission

- HTTP/2 **divides each requested object into smaller units called frames**.
- These frames are **interleaved and multiplexed** over the same TCP connection.
- This means frames from multiple objects can be sent **out-of-order relative to the original requests**, based on priority.
- For example, even if object O1 (large) was requested first, frames of smaller objects O2, O3, and O4 can be sent **first or concurrently**.
- By **interleaving frame transmission**, HTTP/2 greatly reduces HOL blocking at the application layer.

Limitations of HTTP/2 Over TCP

- HTTP/2 still uses a **single TCP connection**.
- TCP guarantees **in-order delivery** of bytes.
- If a TCP segment is lost, **all streams are stalled** until retransmission.
- This creates **HOL blocking at the TCP level**.
- To mitigate this, browsers may open multiple parallel TCP connections.

HTTP/3: Addressing TCP-level HOL Blocking

- HTTP/3 moves to **UDP** transport instead of TCP.
- Implements **per-stream flow control and error recovery**.

- Avoids TCP's HOL blocking by enabling independent retransmission per stream.
- Adds built-in security (encryption).
- Further improves performance and reliability.

📁 File Transfer Protocol (FTP) – Detailed Overview

What is FTP?

- **FTP (File Transfer Protocol)** is a standard protocol used for **transferring files** between a client and a remote server over a network.
- Defined by **RFC 959**.
- It follows the **client-server architecture**:
  - The **client** is the machine or software initiating the file transfer.
  - The **server** is the remote machine that stores files and responds to client requests.
- The FTP server **listens for client connections on TCP port 21**.

How Does FTP Work?

FTP separates control information from data transfer by using **two distinct TCP connections**:

1. Control Connection (Port 21)

- This connection is established first between the FTP client and the FTP server on **TCP port 21**.
- It is used for sending **commands** from client to server, and **responses** from server to client.
- Examples of commands include:
  - Navigating directories (e.g., `cd`, `ls`),
  - Logging in with username and password,
  - Starting a file transfer.
- This connection is **persistent**, remaining open for the duration of the FTP session.
- Called "**out-of-band**" communication because it is separate from the actual file data transfer.

2. Data Connection (Port 20)

- When the client issues a file transfer command (upload or download), the server opens a **second TCP connection** called the **data connection**.
- This connection is typically on **TCP port 20** at the server side.
- The data connection is used solely to **transfer file contents or directory listings**.
- After completing the file transfer, the server **closes the data connection**.

- If more files need to be transferred, new data connections are established as needed.

FTP Session State

- Unlike HTTP, FTP is a **stateful protocol**.
- The FTP server keeps track of:
  - The **current working directory** of the client.
  - The **authentication status** after the client logs in.
  - This allows commands to be interpreted relative to the current directory and enables session continuity.

Summary Diagram

Connection Type	Purpose	Server Port	Characteristics
Control Connection	Commands and server replies	21	Persistent, handles session control
Data Connection	File transfers and listings	20	Opens/closes per transfer, separate from control

---

Why Two Connections?

- Separating control and data allows FTP to handle commands and file transfer independently.
- For example, the client can browse directories or authenticate while a file transfer is occurring on the data connection.
- This separation allows more flexible and interactive sessions compared to protocols using a single connection.

Summary

- FTP uses **two TCP connections**: control (port 21) and data (port 20).
- The control connection manages **commands and session state**.
- The data connection is created dynamically for **file transfer operations**.
- FTP servers maintain session state such as **current directory and authentication**.
- FTP's two-connection model supports interactive file management on remote hosts.

What is Out-of-Band Communication?

Definition

- **Out-of-band (OOB)** communication refers to sending **control information separately** from the main data stream.
- It uses a **different channel or connection** than the one used for the actual data transfer.
- This allows control signals and commands to be sent and received **independently** of data traffic.

In the Context of FTP

- FTP uses **two TCP connections**:
  - **Control connection** (port 21): for sending commands and responses.
  - **Data connection** (port 20): for transferring the actual file data.
- The **control connection is called "out-of-band"** because it operates **separately from the data connection**.
- This separation allows the client to send commands (like listing directories, changing folders, or starting file transfers) **without interfering with the ongoing file data transfer**.

Why is Out-of-Band Important?

- Keeps control messages and data transfers **independent**.
- Improves **flexibility** and **efficiency** of protocols like FTP.
- Enables the client to **manage the session and control the transfer** while data is being sent or received.

Analogy

- Think of a phone call (control connection) and sending a package (data connection) happening simultaneously.
- You can talk on the phone to give instructions (control) while the package (data) is being delivered separately.

FTP Connections: Control Connection vs. Data Connection

---

1. Control Connection

- **Purpose:**
  Used for sending **commands** from the FTP client to the FTP server and receiving **responses** back.
- **How it works:**
  - Established first when the client connects to the server.
  - Uses **TCP port 21** on the server.
  - Remains **open throughout the FTP session**.
  - Commands sent over this connection include: login, change directory, list files, initiate file transfers, etc.
- **Characteristics:**
  - Carries text-based commands and replies.
  - Keeps the session "alive" and maintains the **state** (e.g., current directory, authentication status).
  - Known as the **"out-of-band"** channel because it is separate from the file data transfer.

---

2. Data Connection

- **Purpose:**
  Transfers the **actual file contents** or directory listings between the client and the server.
- **How it works:**
  - Created **only when needed** (e.g., when transferring a file or listing directory contents).
  - Uses **TCP port 20** on the server side (in active mode).
  - Opens **after** the client issues a file transfer command over the control connection.
  - Closes immediately after the transfer completes.
- **Characteristics:**
  - Handles bulk data (file contents, directory listings).
  - Each new file transfer requires opening a **new data connection**.
  - Separate from the control connection to allow commands and data to flow independently.

Summary Table

Feature	Control Connection	Data Connection
Purpose	Sending commands and receiving replies	Transferring file data and directory info

Feature	Control Connection	Data Connection
Server Port	21	20
Connection Lifespan	Persistent during the entire session	Temporary; opens/closes per file transfer
Data Type	Text commands and responses	Binary or text file data
Role in Session	Manages session state and control	Handles actual data transfer

---

Why Separate Connections?

- Separating control and data allows the client to **send commands anytime**, even while transferring data.
- Makes FTP more flexible and interactive compared to protocols that use a single connection for everything.

E-mail System Components and SMTP

Three Major Components of E-mail

1. **User Agents (Mail Readers)**
    - Software that users interact with to **compose, edit, read, and manage e-mails**.
    - Examples: Microsoft Outlook, Apple Mail (iPhone mail client), Gmail web interface.
    - Handles **outgoing and incoming messages**, but does **not directly send or receive messages** from other servers.
    - Outgoing messages are stored in a **message queue** on the user's mail server.
    - Incoming messages are stored in the **user mailbox** on the mail server.
2. **Mail Servers**
    - Store **incoming e-mails** in the user's mailbox until accessed by the user agent.
    - Manage a **message queue** for **outgoing e-mails** that need to be delivered.
    - Communicate with other mail servers to forward e-mails using SMTP.
    - Responsible for relaying messages from the sender's server to the recipient's server.
3. **Simple Mail Transfer Protocol (SMTP)**
    - The **application-layer protocol** used for sending e-mails **between mail servers**.
    - Defines how mail servers communicate to **transfer e-mail messages** across the Internet.
    - SMTP operates in a client-server fashion:
        - The **sending mail server** acts as an SMTP client.
        - The **receiving mail server** acts as an SMTP server.
    - SMTP uses TCP to establish reliable communication between mail servers.

Mail Server Components

Component	Description
User mailbox	Stores incoming e-mails for users until they access them via a user agent.
Outgoing message queue	Stores outgoing e-mails waiting to be sent to other mail servers via SMTP.

Communication Flow

- User composes an e-mail using a **User Agent**.
- The message is sent to the **user's mail server**, which stores it in the **outgoing message queue**.
- The **sending mail server** (SMTP client) establishes a TCP connection to the **receiving mail server** (SMTP server).
- SMTP protocol is used to **transfer the e-mail** from sending to receiving mail server.
- The receiving mail server stores the e-mail in the recipient's **user mailbox**.
- The recipient retrieves e-mails from their mailbox using their user agent (typically via POP3 or IMAP protocols, not covered here).

Visual Summary

```
User Agent --(local interaction)--> User's Mail Server --(SMTP over TCP)--> Recipient's Mail Server -->  Recipient's User Agent
```

Simple Mail Transfer Protocol (SMTP) — RFC 5321

Overview

- SMTP is the **application-layer protocol** used for **reliably transferring email messages** between mail servers.
- SMTP uses **TCP**, typically on **port 25**, to establish a reliable connection.
- The sending mail server acts as the **SMTP client**; the receiving mail server acts as the **SMTP server**.
- Email transfer is usually **direct**: the sending server connects directly to the receiving server.

Three Phases of SMTP Transfer

1. **SMTP Handshaking (Greeting)**
    - TCP connection is established between client (sending server) and server (receiving server).
    - The server responds with a **220 status code** ("Service ready").
    - The client introduces itself with the **HELO** or **EHLO** command.
    - Server replies with **250 OK** if ready.
2. **SMTP Transfer of Messages**
    - The client sends email commands and the message contents using a command/response sequence.
    - Commands and responses are sent as **ASCII text**.
    - Typical commands include:
        - `MAIL FROM:` (sender address)
        - `RCPT TO:` (recipient address)
        - `DATA` (start message content)
    - Server responds with status codes (e.g., 250 for success).
3. **SMTP Closure**
    - After message transfer, the client sends the **QUIT** command.
    - Server closes the TCP connection.

SMTP Command/Response Interaction

- SMTP operates like HTTP with a **command-response protocol**.
- Commands are sent as **ASCII text** strings.
- Server replies with **numeric status codes** plus human-readable phrases.
- Example status codes:
    - `220` — Service ready
    - `250` — Requested action okay
    - `354` — Start mail input (message content)
    - `221` — Service closing transmission channel

Example Scenario: Sending an Email from Alice to Bob

1. **Alice** composes an email in her User Agent, addressed to `bob@someschool.edu`.
2. Alice's mail server (SMTP client) initiates a TCP connection to Bob's mail server (SMTP server) on port 25.
3. SMTP **handshaking**:
    - Bob's server sends `220` greeting.
    - Alice's server replies with `HELO` or `EHLO`.
    - Bob's server responds with `250 OK`.
4. SMTP **transfer**:
    - Alice's server sends `MAIL FROM: alice@example.com`.
    - Bob's server replies `250 OK`.

- Alice's server sends `RCPT TO: bob@someschool.edu`.
- Bob's server replies `250 OK`.
- Alice's server sends `DATA` command.
- Bob's server replies `354 Start mail input`.
- Alice's server sends the message content, ending with a line containing only `.`.
- Bob's server replies `250 OK`.

5. SMTP **closure**:
   - Alice's server sends `QUIT`.
   - Bob's server replies `221 Bye` and closes the connection.

Summary Diagram

```
Client (Alice's Mail Server)                      Server (Bob's Mail Server)
-----------------------                             -------------------------
TCP connection on port 25 ---------------------> TCP connection accepted

Server: 220 Service ready
Client: HELO alice.com
Server: 250 Hello alice.com

Client: MAIL FROM:<alice@example.com>
Server: 250 OK

Client: RCPT TO:<bob@someschool.edu>
Server: 250 OK

Client: DATA
Server: 354 Start mail input

Client: [message content]
Client: .
Server: 250 OK

Client: QUIT
Server: 221 Bye
Connection closed
```

This structure follows the detailed SMTP procedure as described in **RFC 5321** and Kurose & Ross (Application Layer chapter).

End of Message Indication in HTTP vs. SMTP

- **HTTP:**
  - Messages typically end by specifying the **Content-Length** header.
  - The receiver reads exactly that many bytes to know when the message ends.

  - Alternatively, uses chunked transfer encoding to signal message end.
- **SMTP:**
  - Messages end with a line containing only a **single period (.)**.
  - This signals the end of the message data to the receiving server.
  - The server then responds with a status code confirming receipt.

Mail Message Format & Email Retrieval Protocols

Mail Message Format

- **SMTP** (RFC 5321) defines the protocol for exchanging e-mail messages between servers.
- The **email message syntax** itself is defined by **RFC 2822** (similar to how HTML defines web document structure).
- An email message consists of two parts:
  0. **Header**
     - Contains lines like:
       - `To:`
       - `From:`
       - `Subject:`
     - These headers are part of the **message content** and differ from SMTP commands like `MAIL FROM:` or `RCPT TO:`.
  1. **Body**
     - The actual message content, usually ASCII text.
- A **blank line** separates the header from the body.

Retrieving Email: Mail Access Protocols

- **SMTP** is used for **delivering and storing** email messages to the recipient's mail server.
- To **retrieve** emails from the mail server, clients use **mail access protocols**, such as:
  - **IMAP (Internet Mail Access Protocol)** — allows users to retrieve, delete, and manage messages stored on the server. Messages remain on the server.
  - **HTTP** — webmail services (Gmail, Hotmail, Yahoo! Mail) provide a web-based interface for email access, built on top of SMTP and IMAP or POP protocols.

Summary Flow

```
Sender's user agent → SMTP → Sender's mail server → SMTP → Receiver's mail
server
Receiver's user agent → IMAP/HTTP/POP → Receiver's mail server
```

📨 POP3 vs IMAP — Email Retrieval Protocols

📥 POP3 (Post Office Protocol version 3)

- Used to **download emails** from server to client.
- Common mode: **Download and delete** — emails removed from server after download.
- **Limitation:** Emails unavailable on other devices after download.
- Alternative mode: **Download and keep** — emails stay on server.
- POP3 is **stateless** — does **not remember** past sessions.
- Simple, good for offline use.

☁️ IMAP (Internet Message Access Protocol)

- Keeps **all emails on the server**; accessed remotely.
- Supports **folders** to organize emails on server.
- Maintains **user state** across sessions (read/unread, folders).
- Enables consistent access from **multiple devices**.
- Best for modern email with multi-device sync.

🌐 What is DNS? (Domain Name System)

- DNS is like the **Internet's phonebook** — it translates **human-friendly domain names** (like www.google.com) into **IP addresses** (like 142.250.190.78) that computers use to communicate.
- It is a **distributed database** spread across many servers worldwide.
- DNS operates as an **application-layer protocol**, allowing computers and DNS servers to ask and answer queries about domain names.
- This system helps users easily access websites without remembering complex IP addresses.

🌐 DNS: Domain Name System

🔍 Why DNS?

- Humans use many identifiers like SSN, name, passport number.
- Internet devices use **IP addresses** (32-bit) for routing.
- Humans prefer using **names** (e.g., cs.umass.edu) because they are easier to remember.
- **Question:** How do we translate between names and IP addresses (both ways)?

🛠 What is DNS?

- DNS is a **distributed database** made up of a **hierarchy of many name servers**.
- It works as an **application-layer protocol** where hosts and DNS servers communicate to translate between names and IP addresses.
- DNS is a **core Internet function**, but its complexity is managed mostly at the network's edge, not the core.

🧩 DNS Services and Structure

- Why **not centralize** DNS into one server?
  - Would create a **single point of failure**.
  - Would not handle the huge **traffic volume** efficiently.
  - Centralized servers could be **too distant**, increasing query time.
  - Difficult to maintain and update.

📋 Main DNS Services

- **Hostname to IP address translation:** Convert human-readable names to IP addresses.
- **Host aliasing:** Support multiple names for a single host.
  - **Canonical name:** The official name of a host.
  - **Alias name:** Alternative names pointing to the canonical name.
- **Mail server aliasing:** Using different names for mail servers.
- **Load distribution:**
  - Example: One domain name corresponds to multiple IP addresses (e.g., replicated web servers) to balance traffic.

📊 DNS Scale and Volume

- DNS must handle massive query volumes daily, e.g.:
  - Comcast DNS servers: **600 billion queries/day**
  - Akamai DNS servers: **2.2 trillion queries/day**

🌐 DNS: A Distributed, Hierarchical Database

🔍 How DNS Resolves a Domain Name (e.g., www.amazon.com):

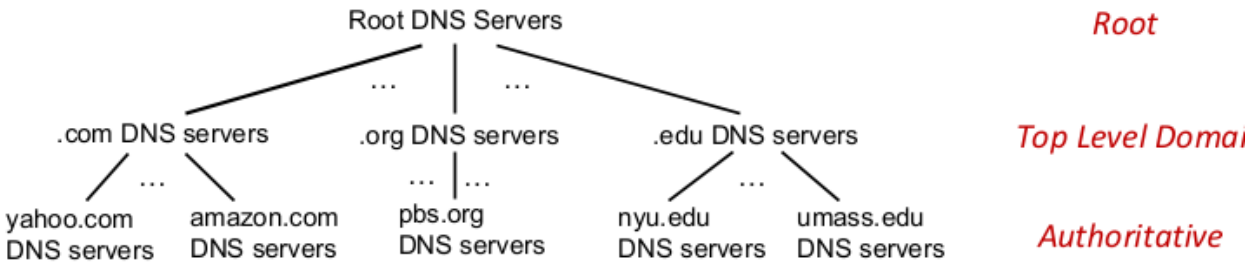
1. The client wants the IP address of **www.amazon.com**.
2. The client first queries a **root DNS server** to find the DNS server responsible for the **.com** top-level domain (TLD).
3. Then, the client queries the **.com DNS server** to get the DNS server responsible for **amazon.com**.
4. Finally, the client queries the **amazon.com DNS server** (authoritative server) to get the IP address for **www.amazon.com**.

🏛️ DNS Hierarchy Overview:

- **Root DNS Servers**: The top of the DNS hierarchy. They know the authoritative servers for all top-level domains (TLDs) like .com, .org, .edu, etc.
- **Top-Level Domain (TLD) DNS Servers**: Handle domain extensions such as `.com`, `.org`, `.edu`, etc.
- **Authoritative DNS Servers**: Hold the actual DNS records for domains like `amazon.com`, `nyu.edu`, `yahoo.com`, etc.

🌳 Hierarchical Structure Example:

```
Root DNS Servers
     |
    .com DNS Servers     .org DNS Servers      .edu DNS Servers
     |                       |                      |
 amazon.com DNS       pbs.org DNS           umass.edu DNS
 DNS Servers          Servers               Servers
     |
 www.amazon.com IP address
```



This distributed hierarchy allows DNS to efficiently resolve domain names worldwide.

🌐 DNS Root Servers, TLDs, Authoritative & Local DNS Servers

🛡️ Root Name Servers

- Root name servers are the **official last resort** for DNS queries that cannot be resolved by other name servers.
- They perform a **critical Internet function**; the Internet would not work without them.
- Managed by **ICANN (Internet Corporation for Assigned Names and Numbers)**.
- There are **13 logical root name servers** worldwide, but each is replicated many times (about 200 servers in the US alone).
- **DNSSEC** adds security to DNS, providing authentication and message integrity.

🏷️ Top-Level Domain (TLD) Servers

- Responsible for top-level domains like:
  - Generic TLDs: `.com`, `.org`, `.net`, `.edu`, `.aero`, `.jobs`, `.museum`, etc.
  - Country-code TLDs: `.cn`, `.uk`, `.fr`, `.ca`, `.jp`, etc.
- Examples of TLD registries:
  - **Network Solutions** manages `.com` and `.net`.
  - **Educause** manages `.edu`.
- TLD servers direct queries to **authoritative DNS servers** for specific organizations.

📌 Authoritative DNS Servers

- Contain the **definitive hostname-to-IP mappings** for an organization's domain.
- Can be managed by the organization itself or by a service provider.
- Respond with the **final answer** for DNS queries related to that domain.

🏠 Local DNS Name Servers

- When a host makes a DNS query, it sends it to its **local DNS server** (usually provided by the ISP).
- Local DNS servers:
  - Answer queries using a **cache** of recent translations (which might be outdated).
  - Forward queries up the DNS hierarchy if the answer is not cached.
- **Local DNS servers don't strictly belong to the DNS hierarchy** but act as a bridge for user queries.

- To find your local DNS server:
  - On MacOS: run `% scutil --dns`
  - On Windows: run `> ipconfig /all`

🔍 DNS Name Resolution: Recursive vs Iterative Queries

When a host (like `engineering.nyu.edu`) wants to find the IP address of a domain (like `gaia.cs.umass.edu`), it asks the DNS system to resolve the name. This can happen in two ways: **recursive query** or **iterative query**.

🌀 Recursive Query

In a **recursive DNS query**, the local DNS server takes full responsibility for resolving the name **on behalf of the client**.

🧠 What Happens?

1. The host sends a query to its **local DNS server** (e.g., `dns.nyu.edu`).
2. If the local DNS doesn't know the answer, it:
   - Queries the **root DNS server**, which directs it to a TLD server.
   - Then queries the **TLD DNS server**, which tells it where to find the authoritative server.
   - Finally, queries the **authoritative DNS server**, which gives the IP address.
3. The **local DNS server returns the final answer** to the host.

☑ **Key Point:** The **client only talks to the local DNS server once**. All other work is done by that DNS server.

📦 Example:

```
Host → Local DNS → (Root → TLD → Authoritative) → Local DNS → Host
```

🔁 Iterative Query

In an **iterative DNS query**, the client does most of the work. The servers respond with a **referral** to another server instead of the final answer.

🧠 What Happens?

1. The host sends a query to the **local DNS server**.
2. The local DNS says: “I don’t know, but try this server” (e.g., root server).
3. The host then asks the **root DNS server**, which refers it to a **TLD server**.

4. Then the host asks the **TLD server**, which refers it to the **authoritative server**.
5. Finally, the host asks the **authoritative server**, which gives the IP address.

☑ **Key Point:** The **client talks to each server in the hierarchy**, following referrals until it gets the final answer.

📦 *Example:*

```
Host → Local DNS → Root → TLD → Authoritative → Host
```

🔁 Iterative DNS Query (Step-by-Step)

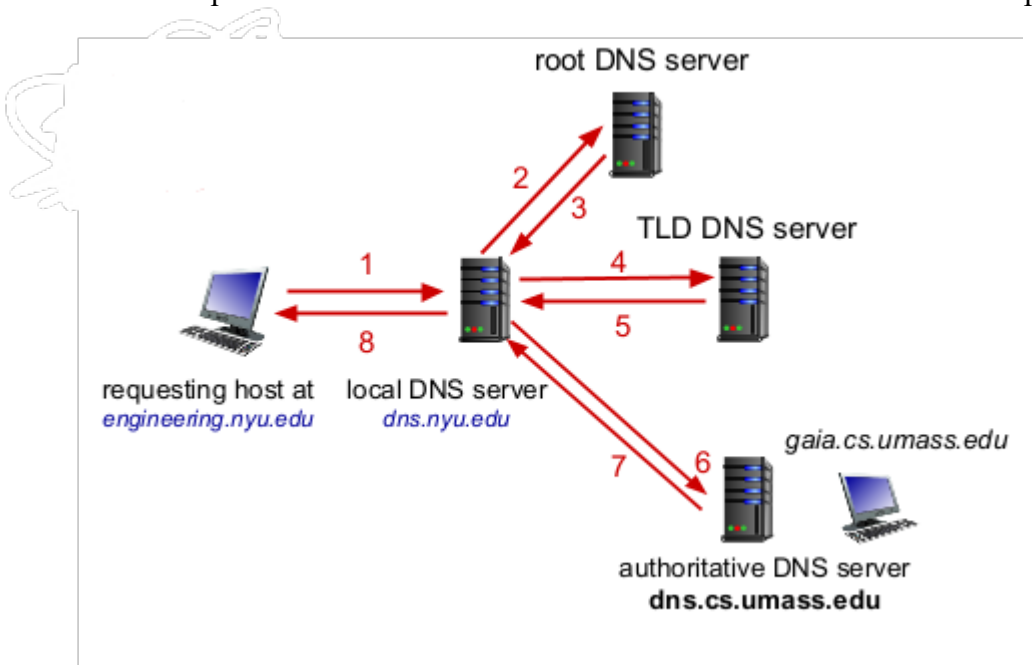
🧠 *Client does more work*

Goal: Host at `engineering.nyu.edu` wants to find IP of `gaia.cs.umass.edu`

📝 *Steps:*

1. 🖥️ **Host** sends request to **local DNS server** (e.g., `dns.nyu.edu`)
2. 🧭 **Local DNS** doesn't know → Asks the **root DNS server**
3. 🧱 **Root server** replies: "I don't know, but ask a TLD server (e.g., `.edu`)"
4. 📩 **Local DNS** asks the **TLD server**
5. 🧱 **TLD server** replies: "I don't know, but ask `cs.umass.edu`'s DNS server"
6. 📩 **Local DNS** asks the **authoritative DNS server** of `cs.umass.edu`
7. ☑ Authoritative server replies with the IP address of `gaia.cs.umass.edu`
8. 📤 Local DNS sends the IP back to the **original host**

🔍 **Key Point:** Each server replies with a **reference** to the next — the local DNS does the step-by-step walk.



🔁 Recursive DNS Query (Step-by-Step)

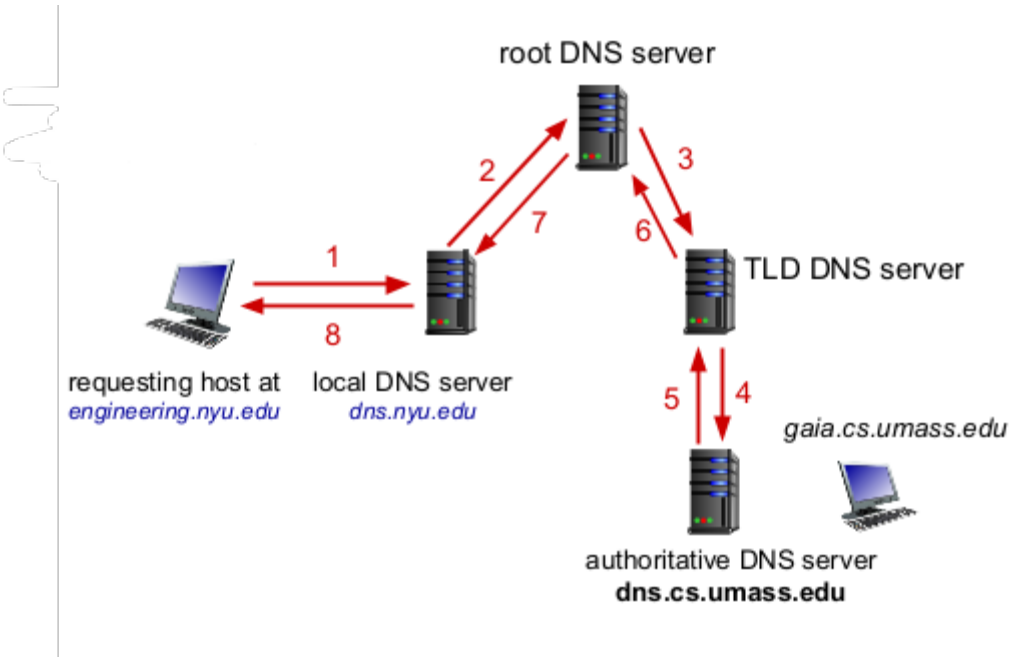
🧠 *Server does more work*

Goal: Host at `engineering.nyu.edu` wants to find IP of `gaia.cs.umass.edu`

Steps:

1. 💻➡️ **Host** sends request to **local DNS server** and says:
   "Please find the IP for me!"
2. 🔄 **Local DNS server** now takes full responsibility.
3. 📩 It queries the **root DNS server**
4. 🧱 Root replies with the TLD DNS server info
5. 📩 Local DNS queries the **TLD server**
6. 🧱 TLD replies with the **authoritative DNS server**
7. 📩 Local DNS queries the **authoritative DNS server**
8. ☑️ Gets the IP of `gaia.cs.umass.edu`
9. 📥 Local DNS returns the IP to the **original host**

🔍 **Key Point:** The **local DNS** does everything and returns final answer — host just waits.



🧠 DNS Caching – Faster Name Resolution

Once any DNS server **learns** a name-to-IP mapping, it **saves (caches)** that info temporarily.

⚙️ How It Works:

1. 🌐 A DNS query is made (e.g., `google.com`)
2. 🗄️ A DNS server finds the IP address from an authoritative source.
3. 💾 It **stores the mapping** (e.g., `google.com → 142.250.190.78`) in a **local cache**
4. 🚀 Next time someone queries the same domain, the DNS server **immediately responds** using the cached value.
5. ⏳ Cached entries have a **time limit**, called **TTL (Time-To-Live)**.

🕒 TTL (Time-To-Live)

- TTL defines **how long** a cached DNS record stays valid.
- After TTL expires, the DNS server **discards** the entry and performs a new lookup.
- Example: TTL = 600 seconds → cache expires after 10 minutes.

⚠️ Warning: Cached Entries May Be Outdated!

- If a domain (like `abc.com`) changes its IP:
  - Old IP may still be used **until TTL expires**.

  - So, not all users see the new IP immediately.
- ➡️ **Best-effort translation** — DNS is not guaranteed to be instantly up-to-date.

📚 DNS Records (Resource Records - RR)

DNS is a **distributed database** that stores information in the form of **Resource Records**.

🔹 **Each RR format**:
`(name, value, type, ttl)`

🔎 Types of DNS Records

1 🅰️ Type A (Address Record)

- 🔤 `name` = hostname
- 🔢 `value` = IP address
- 📌 Used to map a hostname to an IPv4 address
  Example:
  `(www.example.com, 192.0.2.1, A, 3600)`

---

2 🌐 Type NS (Name Server)

- 🔤 `name` = domain (e.g., `example.com`)
- 🧭 `value` = hostname of **authoritative name server** for this domain
  Example:
  `(example.com, ns1.example.com, NS, 86400)`

---

3 🔁 Type CNAME (Canonical Name / Alias)

- 🔤 `name` = alias name
- 👄 `value` = canonical (real) name
- 📌 Used to create an alias for another domain name
  Example:
  `(www.ibm.com, servereast.backup2.ibm.com, CNAME, 86400)`

---

4 Type MX (Mail Exchange)

- 🔤 `name` = domain name
- 📬 `value` = mail server hostname for email delivery
- 📌 Used in email routing

  Example:
  `(example.com, mail.example.com, MX, 7200)`

🌐 Step-by-Step: Getting Your Info into the DNS

Step 1: Choose and Register Your Domain Name

- Company chooses a name like `networkutopia.com`.
- Registers it with a **DNS registrar** (e.g., GoDaddy, Namecheap, Network Solutions).

Step 2: Provide DNS Server Info

- You must provide:
  - **Authoritative Name Server Names** (like `dns1.networkutopia.com`)
  - Their **IP addresses** (e.g., `212.212.212.1`)
- Usually, two servers: **Primary** and **Secondary**

Step 3: Registrar Adds Records to the DNS (TLD Level)

- Registrar puts these records in the **.com TLD server**:

Record Type	Example Entry
`NS` (Name Server)	`networkutopia.com → dns1.networkutopia.com`
`A` (IP Address)	`dns1.networkutopia.com → 212.212.212.1`

Step 4: Create Your Authoritative DNS Server

- Set up your own **authoritative name server** at IP `212.212.212.1`.
- This server will handle DNS queries about your domain.

Step 5: Add More Records

You can now add **custom DNS records** on your authoritative server:

Type	What It Does	Example
`A`	Maps name to IP	`www.networkutopia.com → 212.212.212.1`
`MX`	Mail server	`networkutopia.com → mail.networkutopia.com`

☑ Summary

To get your domain online, you:

1. **Register** a domain.
2. **Provide DNS server info** to registrar.
3. Registrar **adds records** to the TLD server.
4. You **run an authoritative server**.
5. You add more records (like `A`, `MX`) on your server.

⚠ DNS Attacks Explained

The **Domain Name System (DNS)** is like the **Internet's phonebook**, converting domain names (like `www.google.com`) into IP addresses. Since it's so important, attackers often try to exploit or break it. Below are the most common DNS attacks:

💣 1. DDoS Attacks (Distributed Denial of Service)

A DDoS attack tries to **overwhelm DNS servers** with too many requests, causing them to crash or become unresponsive.

---

a) Attack on Root DNS Servers

- **What happens?**
  Attackers flood the **13 root DNS servers** with fake DNS queries.
- **Why?**
  If root servers are down, resolving domain names becomes difficult.
- **Impact?**
  Not much — because **local DNS servers cache TLD info** and **load balancing** is used.

☑ **Defense mechanisms**:

- Anycast routing
- Redundancy
- Caching by local servers

---

b) Attack on TLD Servers

- **What happens?**
  Attackers target **Top-Level Domain servers** like `.com`, `.org`, etc.
- **Why?**
  These servers store mappings of domain names to authoritative DNS servers.
- **Impact?**
  Bigger than root attacks because most domains rely on TLD servers.
- **Result?**
  May cause certain websites to **not resolve** or **slow down**.

---

2. Redirection Attacks (Tricking Users)

These attacks aim to **redirect users to fake or malicious websites** by tampering with DNS responses.

---

a) Man-in-the-Middle Attack (MITM)

- **What happens?**
  A hacker places themselves between your device and DNS server.
- **Goal:**
  Intercept and modify DNS responses to redirect you.
- **Result:**
  You visit a fake site thinking it's real (like a fake bank site).

---

b) DNS Cache Poisoning

- **What happens?**
  Attacker sends **false DNS replies** to a DNS server **before** the real one arrives.
- **Goal:**
  The server stores (caches) the wrong IP address.

- **Impact:**
  Every user who queries that domain gets sent to the wrong (often malicious) site.

🧠 **Example**:

- Attacker poisons the cache for `facebook.com` → redirects to a fake login page.

🔬 3. DNS as a Tool for DDoS (Amplification Attack)

DNS can also be **used by attackers to attack other systems**.

- **How?**
  The attacker sends a DNS request using a **spoofed IP** (the victim's IP).
- **What happens?**
  The DNS server replies to the **victim**, not the attacker.
- **Why is it powerful?**
  A small request can produce a **much larger reply**, overloading the victim's system.

🔁 **Amplification Example**:

- A 60-byte query generates a 4000-byte response.
- This multiplies the attack power **many times**.

🎥 Video Streaming and CDNs: Easy Explanation

📌 *What is Video Streaming?*

Video streaming means watching videos **directly from the internet** without downloading the whole file first.

Examples:

- Watching Netflix, YouTube, Amazon Prime, etc.

💡 Why Is It Challenging?

1. Scale (Huge Number of Users)

- Platforms like Netflix or YouTube serve **millions to billions** of users at once.
- If all users tried to stream from one central server, it would crash due to overload.

2. Heterogeneity (Differences Among Users)

- Not all users have the same internet quality or devices.
  - Some use fast fiber internet; others use slow mobile data.
  - Some use 4K TVs; others use small phone screens.
- So, the video service must adapt to each user's condition (called **adaptive streaming**).

☑ The Solution: CDNs (Content Delivery Networks)

A **CDN** is a **network of many servers placed in different locations around the world**. These servers **store copies of videos** and deliver them to users based on proximity.

How it works:

1. **User opens Netflix and clicks on a video.**
2. Instead of getting the video from a central server in the US (which is far away), Netflix finds the **closest server** to the user (maybe in Dhaka or Kolkata).
3. That nearby CDN server sends the video **quickly and smoothly**.

📦 Benefits of CDNs

- **Faster video loading** (low buffering)
- **Less delay** (low latency)
- **Handles large traffic** by distributing the load across servers
- **Delivers best quality possible** based on user's device & internet speed

🎬 Multimedia: Video

What is a Video?

A **video** is simply a **sequence of images** (called *frames*) that are shown **one after another** very quickly.

- Example: 24 images per second = **24 frames/sec**
- These images create the illusion of **motion** when viewed continuously.

🧩 What is a Digital Image?

Each image is made up of tiny dots called **pixels**.

- Every pixel has a **color value**, which is represented using **bits** (like binary numbers).
- More bits = better quality (but also larger size).

Coding (Compression)

To reduce file size, we use **coding techniques** to **avoid sending unnecessary data**.

There are **two main types of coding:**

---

1. Spatial Coding (within a single image)

- Many parts of an image may have the **same color repeated**.
- Instead of sending each pixel one by one, we send:
  - The **color**
  - The **number of times it repeats**

Example:

If a row in an image has:

```
text
CopyEdit
Purple, Purple, Purple, Purple, Purple
```

➞ Instead of sending 5 times "Purple", we send:

```
text
CopyEdit
(Purple, 5)
```

This saves bits and reduces file size.

---

2. Temporal Coding (between images)

- In a video, **many frames are similar** to the previous one.
- Instead of sending the entire next frame, we just send the **difference** from the last frame.

Example:

Let's say:

- Frame `i` and frame `i+1` are 90% the same.
- We only send the **10% that changed** in frame `i+1`.

🟩 This saves a lot of data during video playback.

🎥 Multimedia: Video

🧩 Spatial Coding (within a single image)

- When many pixels have the **same color**, instead of sending each pixel separately, send only **two values**:
    0. The **color value** (e.g., purple)
    1. The **number of times** that color repeats (N)

Example:
Instead of sending `Purple, Purple, Purple, Purple` (4 times),
send `(Purple, 4)` — saves space!

⏩ Temporal Coding (between frames)

- Instead of sending the **entire next frame**, send only the **differences** compared to the previous frame.

Example:
Frame i and frame i+1 are mostly similar,
so send just what changed from frame i to frame i+1.

📊 Bit Rate Types in Video Encoding

- **CBR (Constant Bit Rate):**
  The video is encoded at a **fixed bit rate** regardless of complexity.
- **VBR (Variable Bit Rate):**
  The bit rate **changes dynamically** depending on how much data needs to be encoded (more complex scenes = higher bit rate).

Examples of Video Encoding Standards

Standard	Typical Bit Rate	Usage
MPEG-1	1.5 Mbps	CD-ROM video
MPEG-2	3 - 6 Mbps	DVDs
MPEG-4	64 Kbps – 12 Mbps	Internet streaming

Streaming Stored Video

Scenario Overview:

- A **video server** stores the entire video file.
- A **client** (like your laptop, phone, or smart TV) requests the video from the server over the Internet.
- The video is **streamed** — meaning the client plays the video as it receives data, without waiting for the entire file to download first.

Main Challenges in Streaming Stored Video:

1. **Variable Bandwidth Availability:**
    - The available network bandwidth between the server and client **changes constantly**.
    - Causes include:
        - **Local network congestion:** e.g., multiple devices in your home using internet at once.
        - **ISP access network congestion:** many users sharing the same connection point.
        - **Core Internet congestion:** traffic passing through routers and data centers.
        - **Server load:** many clients requesting video simultaneously can slow the server.
2. **Packet Loss and Delay:**
    - Due to congestion or other issues, some data packets can be **lost or delayed**.
    - Lost packets cause the video player to **stall or buffer**, interrupting smooth playback.
    - Delays can also reduce video quality if the player compensates by lowering the video bitrate.
3. **Impact on Video Playback:**
    - These network problems may cause:
        - **Buffering pauses**, where playback temporarily stops to load more data.

- **Reduced video quality** (lower resolution or bitrate) to maintain continuous playback.
- **Unpleasant user experience** if the video frequently stalls or looks blurry.

How Streaming Adapts (Intro Preview):

To handle these challenges, modern streaming uses techniques like **adaptive bitrate streaming**, where the video quality changes dynamically to match the current network conditions. This helps minimize buffering and keeps playback smooth.

📺 Streaming Stored Video Explained

---

1. Basic Process of Streaming Stored Video

- A video is **recorded** at a certain frame rate (e.g., 30 frames per second).
- The **video data is sent** from the server to the client over the network.
- While the server is still sending the later parts of the video, the client **starts playing the earlier parts**.
- This process is called **streaming** — you can watch the video as it's being downloaded.

---

2. Network Delay

- There is a **network delay** — the time it takes for data to travel from server to client.
- In this example, the delay is **fixed**, but in reality, it often varies.

⚠️ Challenges in Streaming Stored Video

---

a) Continuous Playout Constraint

- The video on the client must be played **continuously and smoothly**, keeping the original timing.
- For example, if the video is 30 frames/sec, the client should play frames exactly that fast.
- However, because **network delays vary** over time (called *jitter*), the client can't always get data exactly on time.
- To handle this, the client uses a **buffer** — it temporarily stores some video data before starting playback, so playback stays smooth despite network fluctuations.

---

b) Client Interactivity

- Users want to **pause, fast-forward, rewind, or jump** through the video.
- This requires special handling because the client may need to request different parts of the video quickly.

---

c) Packet Loss and Retransmission

- Video data sent over the network may get lost or corrupted.
- Lost packets may be **retransmitted**, which can cause delays or interruptions in playback.

🎬 Playout Buffering Explained

- The video is typically sent at a **constant bit rate** (CBR), meaning data flows at a steady speed.
- But the network delay varies, so the **client's received data** can be irregular.
- To keep playback smooth, the client:
    0. **Buffers** (stores) some amount of video data before starting playback.
    1. **Delays playback start** slightly to fill this buffer.
    2. During playback, continues to use the buffer to compensate for network delay changes (jitter).

📊 Summary Flow:

Step	Description
1	Server sends video data at constant rate
2	Network causes variable delay in data arrival
3	Client buffers received data before playing
4	Client plays video at original frame rate (e.g., 30 fps)
5	Buffer absorbs delay variations to keep playback smooth

In simple words:
Streaming video means the client is watching while still downloading. Because networks are unpredictable, the client stores some video in advance (buffering) to avoid pauses or glitches in playback.

🎥 What is DASH?

Dynamic Adaptive Streaming over HTTP (DASH) is a modern video streaming technique designed to deliver smooth and high-quality video over the Internet, even when network conditions vary.

🌐 Why DASH?

Internet bandwidth can change frequently due to network congestion, user mobility (e.g., switching from Wi-Fi to mobile data), or other factors. DASH adapts the video quality on the fly to provide the best possible viewing experience without interruptions or buffering delays.

⚙️ How DASH Works:

1. Server Side Preparation:

- The original video file is **divided into small chunks or segments** (usually a few seconds long).
- Each chunk is **encoded multiple times at different quality levels or bitrates** (e.g., low, medium, high).
- These multiple versions are stored as separate files.
- These files are **distributed across many CDN (Content Delivery Network) servers** globally to bring content closer to users.
- A **manifest file (or MPD – Media Presentation Description)** is created, listing the URLs of all video chunks and their quality versions. This manifest helps the client know what chunks are available and where to get them.

2. Client Side Playback:

- The client periodically **measures the current network bandwidth** and other conditions.
- Using the manifest file, the client **requests one video chunk at a time**.
- For each chunk, the client **selects the highest-quality chunk that can be downloaded smoothly** given the current bandwidth.
- If bandwidth improves, the client can switch to a higher-quality chunk for better video clarity.
- If bandwidth drops, the client switches to a lower-quality chunk to avoid buffering.

- The client can also request chunks from **different CDN servers** to optimize delivery speed and reliability.

🔑 Key Benefits of DASH:

- **Adaptive Quality:** Video quality changes dynamically based on network conditions, ensuring continuous playback without pauses.
- **Efficient Bandwidth Use:** Uses available bandwidth optimally without overloading the network.
- **Scalability:** Works well for large audiences because it uses HTTP and standard CDN infrastructure.
- **Client-Controlled:** The client controls which chunk quality to download next, making it responsive to local conditions.

📊 Visual Summary:

Step	Description
1	Video divided into small chunks, encoded at different qualities
2	Manifest file lists all chunk URLs and qualities
3	Client measures bandwidth and requests chunks one by one
4	Client selects chunk quality based on bandwidth, switching dynamically
5	Video plays smoothly with minimal buffering and best possible quality

🤖 Streaming Multimedia: DASH Client Intelligence Explained

In **Dynamic Adaptive Streaming over HTTP (DASH)**, most of the smart decisions happen at the **client side**, which helps deliver smooth video playback even with changing network conditions.

The client handles three main decisions:

---

1. When to Request a Video Chunk

- The client continuously monitors its **playout buffer** — the temporary storage holding video data before it's played.
- It requests the next chunk **at the right time** to avoid:
  - **Buffer starvation**: Running out of video data to play, causing pauses or freezing.
  - **Buffer overflow**: Downloading too much data too soon, wasting bandwidth and memory.
- This timing ensures a **steady flow** of video data for uninterrupted playback.

---

2. What Encoding Rate to Request

- Each video chunk is available in multiple quality levels (different bitrates).
- The client **measures current network bandwidth** periodically.
- If bandwidth is high, it requests **higher quality chunks** for better video clarity.
- If bandwidth drops, it switches to **lower quality chunks** to avoid buffering delays.
- This adaptive choice ensures the best possible video quality **without interruptions**.

---

3. Where to Request the Chunk From

- Video chunks are stored on multiple servers, often across a **Content Delivery Network (CDN)**.
- The client selects the **server (URL) that can deliver the chunk fastest**, which might be:
  - The server geographically closest to the client.
  - The server with the least network congestion or highest available bandwidth at that moment.
- This helps minimize delays and improves playback smoothness.

🎬 Putting It All Together

- Streaming video smoothly involves three components working together:
  - **Video encoding**: compressing video into multiple quality levels.
  - **DASH streaming**: adaptive chunk-based delivery of video.
  - **Client-side playout buffering**: buffering enough data to handle network variability.
- The client's intelligent management of chunk requests based on timing, quality, and source is key to providing a seamless viewing experience.