Protocols

Contents

I. Protocol Design and Properties 8

1. What is a Protocol? 8

Why Protocols Matter 8

2. Key Protocol Properties 8

Data Format 8

Transfer Mode 9

Addressing System 9

Directionality 10

State Management 10

Routing & Proxies 11

Flow & Congestion Control 11

Error Management 11

II. Understanding the OSI Model 12

1. Why This Matters 12

2. What is the OSI Model? 12

3. The 7 Layers of the OSI Model 13

7. Application Layer 13

6. Presentation Layer 13

5. Session Layer 13

4. Transport Layer 14

3. Network Layer 14

2. Data Link Layer 14

1. Physical Layer 14
Flow of a Request (Client to Server) 14
Server Receives: 16
How Devices Use OSI Layers 16
4. TCP/IP Model 18
Terminology Recap 18
III. Internet Protocol 19
1. What is the Internet Protocol (IP)? 19
2. IP Address Basics 19
Network and Host Portion 20
Subnet Mask 21
3. Default Gateway 22
4. Anatomy of the IP Packet 23
5. Fragmentation 24
6. Time To Live (TTL) 24
What is ICMP? 24
7. Protocol Field 25
8. Ping & Traceroute 26
IV. UDP 27
1. Characteristics of UDP 27
2. Use Cases of UDP 27
3. UDP Internals: Ports and Multiplexing 28
4. UDP Packet Structure (Datagram Anatomy) 28
5. Pros and Cons of UDP 29

6. Security Considerations 29
V. TCP 30
1. Key Features of TCP 30
2. TCP vs. UDP 31
3. TCP Segment Structure 31
Flags Overview 32
4. Connection Establishment (3-Way Handshake) 33
5. Data Transmission 33
6. Memory and Resource Usage 33
7. Connection Termination (4-Way Handshake) 34
8. Maximum Segment Size (MSS) 34
9. TCP Under Load: Connection Queue vs Memory Buffers 34
Connection Queue (Before Connection Established) 34
Receive Buffer (After Connection Established) 36
VI. Transport Layer Security 37
Where TLS Fits in the OSI Model 37
1. HTTP over TCP 37
Connection Termination 38
Port Details 38
Summary Table 39
2. HTTPS Communication (HTTP + TLS + TCP) 39
A. TCP 3-Way Handshake (Same as HTTP) 39
B. TLS Handshake (Before HTTP Starts) 40
C. HTTP Request (Encrypted Inside TLS) 43

D. HTTP Response (Encrypted Too) 43
E. Connection Termination 43
3. Why Not Use RSA for Everything? 43
4. Certificates and Authentication 43
VII. HTTP/1.1 44
1. Structure of an HTTP Request 44
a. Method: 44
b. Path 45
c. Protocol 45
d. Headers (Key-Value Pairs) 45
e. Body (Optional) 46
2. Structure of an HTTP Response 46
a. Protocol 46
b. Status Code 46
c. Status Text 47
d. Headers 47
e. Body 48
3. HTTP 1.0: The Original Approach 48
4. HTTP 1.1: Improvements and Persistent Connections 49
5. HTTP/2 Overview 51
6. HTTP/3 (Over QUIC) 51
7. Summary 52
VIII. HTTPS, TLS, and Encryption 53
1. HTTP and HTTPS Fundamentals 53

2. Encryption Types 54
Symmetric Encryption 54
Asymmetric Encryption 54
3. Certificates 55
4. Certificate Authorities (CAs) 56
5. Certificate Verification: 56
6. TLS Handshake (Simplified) 57
IX. WebSockets 58
1. Why WebSockets? The Limitations of HTTP 58
2. How WebSockets Work 58
3. Use Cases for WebSockets 60
X. HTTP/2 61
1. HTTP/1.1 Refresher 61
2. HTTP/2 Introduction 61
3. Multiplexing over a Single Connection: 62
4. Header Compression: 62
5. Server Push (Deprecated/Replaced): 62
6. Secure by Default (Practical Enforcement): 63
7. Pros and Cons of HTTP/2 63
Pros: 63
Cons: 64
XI. Many ways to HPPTS 65
1. The Three Pillars of HTTPS Communication 65
2. Variants of HTTPS Communication 66

3. HTTPS: The Secure Way to Communicate 66
TLS 1.2: The Minimum Standard for Security 66
4. HTTPS Over TCP with TLS 1.2 67
Phase 1: The TCP Three-Way Handshake (Establishing the Foundation) 67
Phase 2: The TLS Handshake (Securing the Connection with Encryption) 68
Data Transfer (Encrypted Communication) 70
Summary: TLS 1.2 Workflow 70
5. HTTPS Over TCP with TLS 1.3 71
Layers Involved 71
Phase 1: The TCP Three-Way Handshake (Foundation Remains) 71
Phase 2: The TLS 1.3 Handshake (Optimized for Speed) 71
Encrypted Data 73
Quick Recap 74
6. HTTPS over TCP Fast Open (TFO) 75
The Prerequisite for 0-RTT 75
Phase 1: The TCP Three-Way Handshake (Standard) 75
Phase 2: The TLS 1.3 0-RTT Handshake (Accelerated) 76
7. TLS 1.2 RSA vs TLS 1.2 vs TLS 1.3. 77
XII. Backend Execution Patterns 78
1. How TCP Connections Are Established 78
TCP Connection Establishment: The 3-Way Handshake 78
Server Listening: Address and Port Explained 78
The Kernel's Role in Connection Management 79
The Kernel's Connection Queues 80

Application's Role: The `accept()` Call 82
2. Reading and Writing Data on TCP Connections 83
Understanding Send and Receive Buffers 83
How Data is Received by the Server (Client Sends to Server) 83
How the Application Reads Data (Server Reads from Receive Buffer) 85
How Data is Transmitted by the Server (Server Sends to Client) 86
3. Listener, Acceptor, and Reader 87
The Listener 87
The Acceptor 88
The Reader 88
Single Listener, Single Worker Thread 88
Single Listener + Multiple Reader Threads 89
Single Listener with Request-Level Load Balancing 90
Multiple Threads with a Single Socket 93
Multiple Listeners on the Same Port (Socket Sharding) 94

I. Protocol Design and Properties

1. What is a Protocol?

- A communication protocol is a **system of rules** that allows two parties to communicate. A **protocol** is a **set of rules** that both sender and receiver follow to exchange data correctly.
    - Example: Just like speaking the same language lets two people talk, following the same protocol allows computers to communicate.

Why Protocols Matter

- Every protocol is designed to solve a specific problem.
- For example:
    - TCP was designed in the 1960s for low-bandwidth networks.
    - Today, in high-speed data centres, TCP is hitting its limits, so newer protocols like **Homa (2022)** are being explored.

2. Key Protocol Properties

- These are the **aspects you may consider** when designing or working with protocols.

Data Format

- How is the data structured when sent?

Text-based:

- Human-readable (if not encrypted).
- Examples: **HTTP** (plain text), **JSON**, **XML**

Binary:

- Optimized for machine processing, not human reading.
- Examples: **gRPC**, **Protocol Buffers**, **RESP** (used by Redis)
- More efficient for performance-critical applications.

Transfer Mode

- How data is transmitted across the network.

Message-Based:

- Has a clear start and end.
- Each message is a distinct unit.
- Examples:
    - UDP
    - HTTP (when considered at the message level)

Stream-Based:

- Just a continuous flow of bytes, no defined boundaries.
- It's up to the application to understand message limits.
- Examples:
    - TCP
    - RTSP (used in video streaming)
    - Note: When HTTP is built over TCP, the client must parse the stream to detect message boundaries.

Addressing System

- Every protocol must know where the data is going.
- Types of Addressing:

Layer	Example
Layer 2 (Data Link)	**MAC address**
Layer 3 (Network)	**IP address (e.g., 192.168.1.1)**
Layer 7 (Application)	**DNS name (e.g., www.google.com)**

- **DNS** resolves a domain name to an **IP** (You type www.google.com → DNS gives you 142.250.182.68.), which is eventually mapped to a **MAC address** using **ARP** (IP: 192.168.1.10 → ARP → MAC: 00-1A-2B-3C-4D-5E).
    - ARP (Address Resolution Protocol)

- Once the IP address is known, the system still needs to send data over the local network.
- To do that, it needs the MAC address (hardware address) of the destination.
- ARP is used to resolve an IP address to a MAC address on the local network.

Directionality

- How communication flows:

Mode	Description
Unidirectional	One-way only
Bidirectional	Both parties can send and receive
Half-Duplex	Only one party can send at a time (e.g., WiFi)
Full-Duplex	Both parties can send and receive simultaneously

Why is Wi-Fi half-duplex?

- On a Wi-Fi network, if your laptop is sending data, it must finish before it can receive data from the router. The transmission happens in **turns**.
- Wi-Fi uses radio frequencies that are shared across devices on the same channel.
- If two devices transmit at the same time, it causes collisions.
- So, Wi-Fi uses a protocol called CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance) to take turns sending data.

State Management

- How much memory (state) the protocol needs to track ongoing communication.

Type	Example
Stateful	**TCP, gRPC, FTP** — tracks connection state
Stateless	**UDP, HTTP (in theory)** — no memory of past messages

Routing & Proxies

- How messages pass through the network and possibly through intermediate proxies.
- Example:
    - Client sends request to google.com.
    - DNS resolves to an IP address.
    - TCP connects to a proxy (not directly to Google).
    - Proxy forwards request to Google.
    - Response is sent back through the same route.
    - Protocols must support such intermediaries and handle routing correctly.

Flow & Congestion Control

- Some protocols monitor traffic to avoid overwhelming the network.

Feature	In TCP	In UDP
Reliable Delivery	✅ Yes	❌ No
Congestion Control	✅ Yes	❌ No
Flow Control	✅ Yes	❌ No
Retransmission of Lost Packets	✅ Yes	❌ No

Error Management

- What happens when something goes wrong?
- Includes:
    - Error codes (e.g., 404, 500)
    - Timeout handling
    - Retry strategies
    - Protocols like HTTP define standard error handling rules.
    - Some (like UDP) leave error handling up to the application.

II. Understanding the OSI Model

1. Why This Matters

- Many developers ignore the OSI model early on, focusing only on application development.
- But understanding the OSI model is crucial when building, debugging, or scaling networked applications.
- It helps you:
    - Understand where your app fits in the network stack.
    - Diagnose issues (e.g., DNS, TCP failures).
    - Design better APIs, proxies, gateways, and services.

2. What is the OSI Model?

- Definition: **A standardized conceptual model with seven layers, each defining a specific function in data communication across networks**.
- It enables:
    - Interoperability between systems
    - Modular development and innovation
    - Network troubleshooting and planning
- Goals of the OSI Model
    - Build agnostic applications that work regardless of network type (Wi-Fi, Ethernet, Fiber, LTE).
    - Enable interoperability: different vendors, protocols, devices work together.
    - Decouple hardware from software logic.
    - Allow network upgrades without breaking application logic.
    - Enable independent innovation at each layer.

3. The 7 Layers of the OSI Model

- Each layer wraps data with additional info for the layer below (like nested boxes or **Russian dolls**).

Layer	Name	Role	Examples
7	Application	Interface for end-user	HTTP, FTP, SMTP
6	Presentation	Data formatting, encoding	JSON, XML, TLS, JPEG
5	Session	Connection state, TLS handshake	TLS, gRPC sessions
4	Transport	Reliable/unreliable delivery, ports	TCP, UDP, QUIC
3	Network	Logical addressing, routing	IP, ICMP
2	Data Link	Physical addressing (MAC)	Ethernet, Wi-Fi
1	Physical	Transmission medium	Fiber, radio, electrical

7. Application Layer

- Closest to the user.
- Interfaces with the actual **web/app logic**.
- Examples: **HTTP (Express), FTP clients, WebSocket apps**

6. Presentation Layer

- Handles **serialization, encoding, encryption**.
- Translates data into a format the app understands.
- Examples: **JSON.stringify, UTF-8 encoding, TLS encryption**

5. Session Layer

- Manages connections and sessions.
- Tracks state (e.g., for **gRPC**, **database pooling**).

- Important in **TLS** (**handshakes**, **session caching**).
- Examples: **TLS**, **Netty**, **Linkerd**

4. Transport Layer

- Provides **end-to-end communication**, port management, and reliability.
- Distinguishes between applications on a device using **ports**.
- Examples:
    - **TCP**: Reliable, ordered, with retransmissions.
    - **UDP**: Faster, no guarantee, used in real-time apps.
    - **QUIC**: Modern protocol over UDP.

3. Network Layer

- Handles **routing** via **IP addresses**.
- Sends **packets** between devices across networks.
- Example: **IP (IPv4/IPv6), ICMP**

2. Data Link Layer

- Deals with **frames** and **MAC addresses**.
- Ensures data is sent to the **right physical device**.
- Examples: **Ethernet, ARP, Wi-Fi (802.11)**

1. Physical Layer

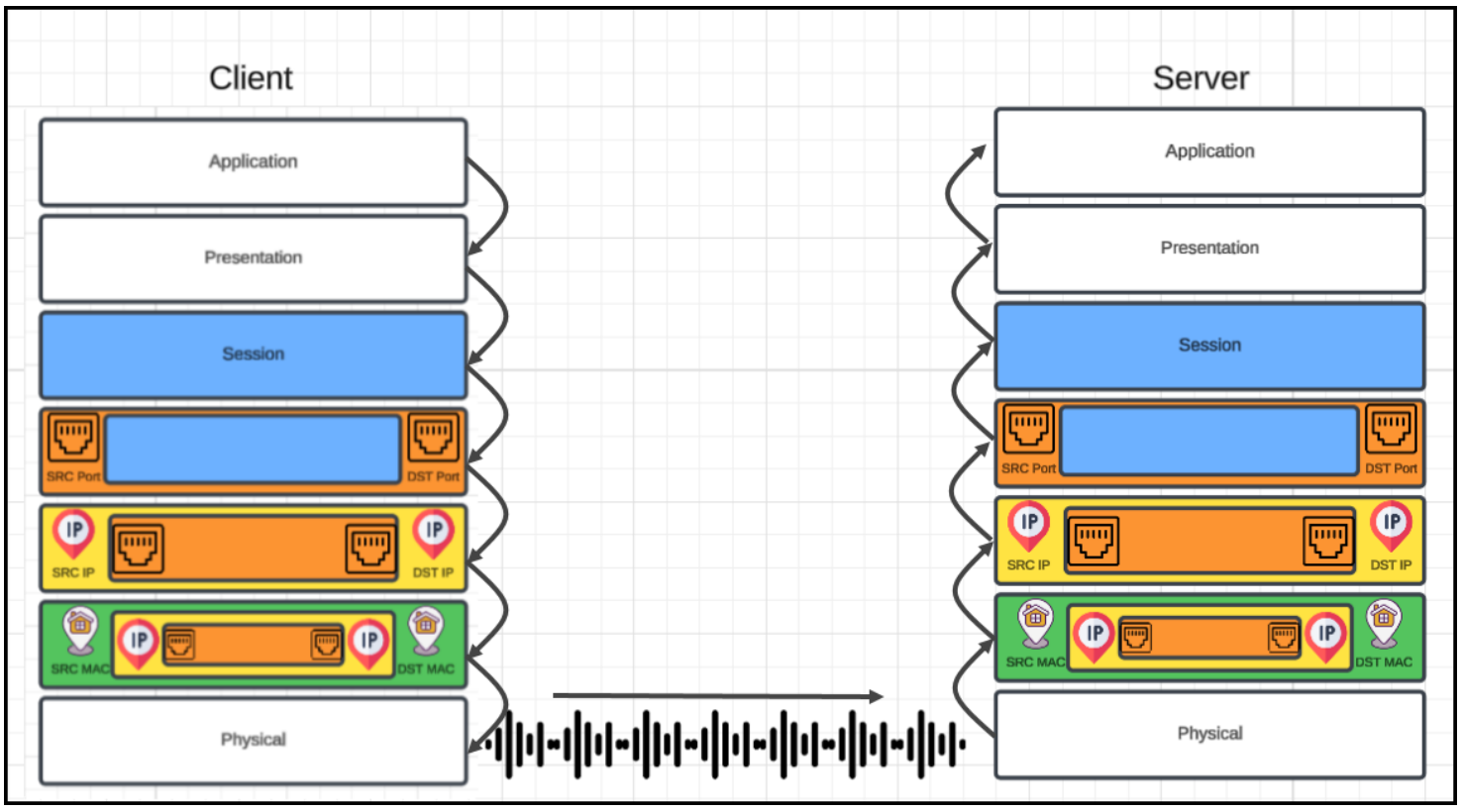
- Deals with **electrical/radio/light signals**.
- Converts digital bits to signals for transmission.
- Examples: **Ethernet cables, Fiber optics, Wi-Fi radio**

Flow of a Request (Client to Server)

Example sending a POST request to an HTTPS webpage

- Layer 7 - Application
    - POST request with JSON data to HTTPS server.
- Layer 6 - Presentation
    - Serialize JSON to flat byte strings.

- Layer 5 - Session
    - Request to establish TCP connection/TLS.
- Layer 4 - Transport
    - Sends SYN request target port 443.
- Layer 3 - Network
    - SYN is placed an IP packet(s) and adds the source/destination Ips.
- Layer 2 - Data link
    - Each packet goes into a single frame and adds the source/ destination MAC addresses.
- Layer 1 - Physical
    - Each frame becomes string of bits which converted into either a radio signal (Wi-Fi), electric signal (ethernet), or light (Fiber).



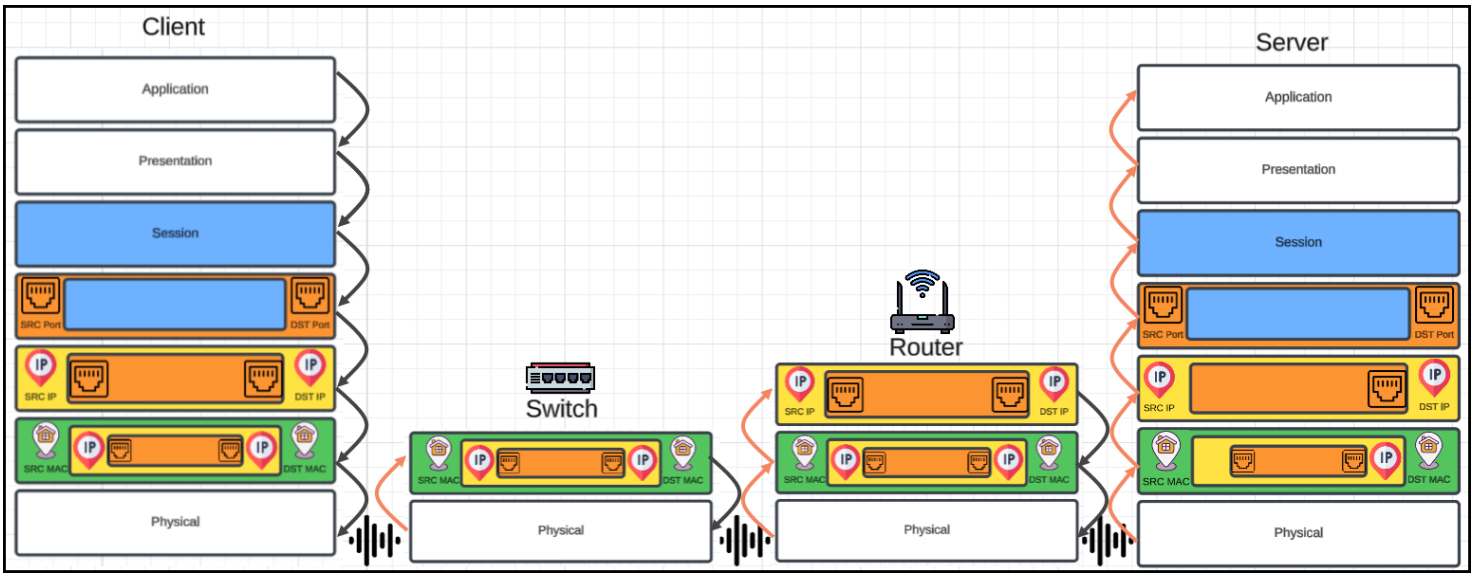
Server Receives:

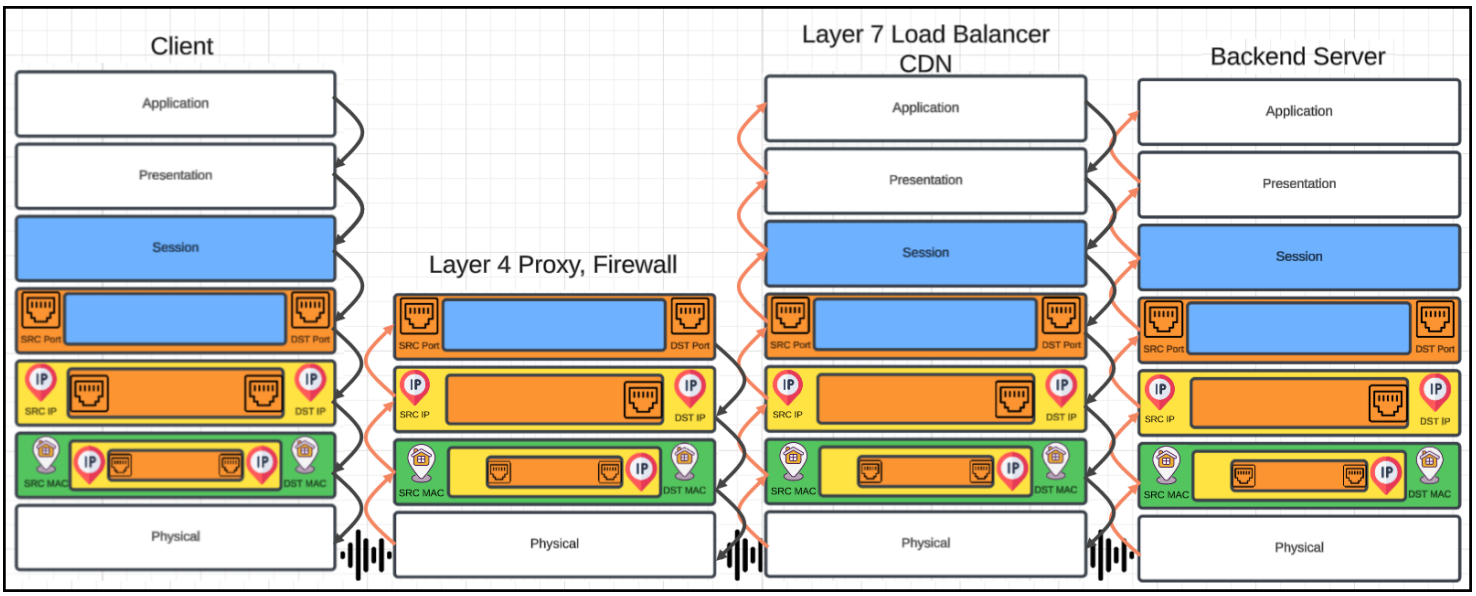
Receiver computer receives the POST request the other way around

- Layer 1-Physical
    - Radio, electric or light is received and converted into digital bits.
- Layer 2-Data link
    - The bits from Layer 1 are assembled into frames.
- Layer 3-Network
    - The frames from layer 2 are assembled into IP packet.
- Layer 4-Transport
    - The IP packets from layer 3 are assembled into TCP segments.
    - Deals with Congestion control/flow control/retransmission in case of TCP.
    - If Segment is SYN we don't need to go further into more layers as we are still processing the connection request.
- Layer 5-Session
    - The connection session is established or identified.
    - We only arrive at this layer when necessary (three-way handshake is done).
- Layer 6-Presentation
    - Deserialize flat byte strings back to JSON for the app to consume.
- Layer 7-Application
    - Application understands the JSON POST request and your express Json or Apache request receive event is triggered.

How Devices Use OSI Layers

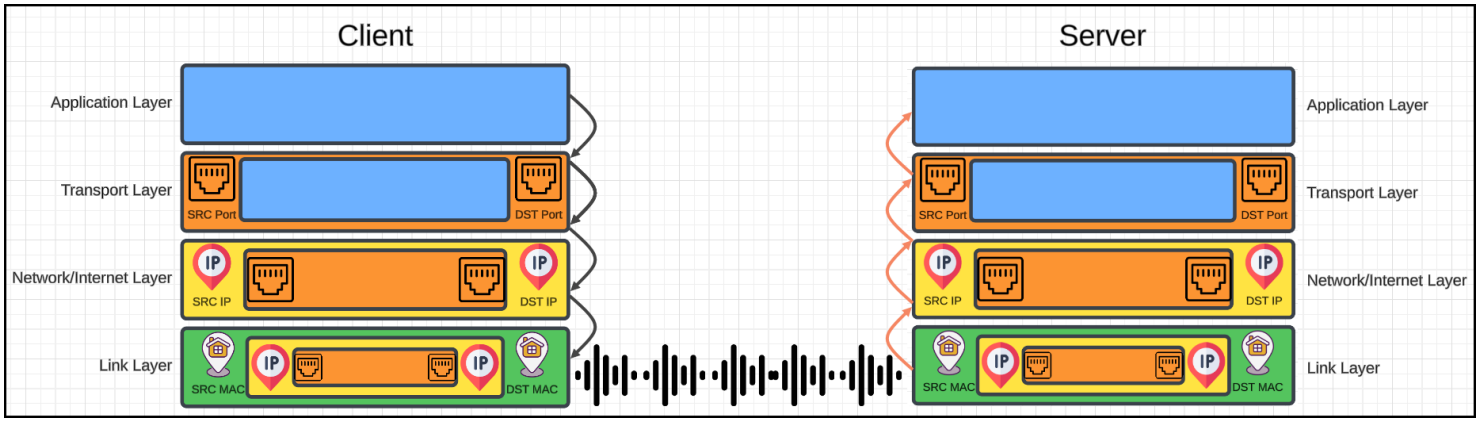
- Switches work at Layer 2 (MAC/frame).
- Routers work at Layer 3 (IP/packet).
- Firewalls may inspect Layer 4 or even Layer 7.
- Reverse Proxies / CDNs work at Layer 7 (HTTP, gRPC).
- Load Balancers might operate at Layer 4 (ports) or 7 (URLs).


Client
Application
Presentation
Session
SRC Port
DST Port
IP
SRC IP
DST IP
SRC MAC
DST MAC
Physical
Switch
Router
Server


Client
Application
Presentation
Session
SRC Port
DST Port
IP
SRC IP
DST IP
SRC MAC
DST MAC
Physical
Layer 4 Proxy, Firewall
Layer 7 Load Balancer
CDN
Backend Server

4. TCP/IP Model

- Much simpler than OSI just 4 layers
- Application (Layer 5, 6 and 7)
- Transport (Layer 4)
- Internet (Layer 3)
- Data link (Layer 2)
- Physical layer is not officially covered in the model



Terminology Recap

Layer	**Unit Name**	**Example Term**
1	Bits	1s and 0s
2	Frames	Ethernet frame
3	Packets	IP packet
4	Segments	TCP segment
4	Datagrams	UDP datagram

III. Internet Protocol

1. What is the Internet Protocol (IP)?

- IP is the "vehicle" of the internet—it delivers data between machines over a network.
- Regardless of the protocol used (**HTTP**, **gRPC**, **UDP**, **TCP**, **database responses**, etc.), all data is ultimately wrapped inside an **IP Packet**.
- IP Packet (Layer 3) Contains:
    - Source IP address (where data is from)
    - Destination IP address (where data should go)
- Routers only care about **IP packets**—not ports, encryption, or HTTP headers.

2. IP Address Basics

- Belongs to Layer 3 of the OSI Model.
- Format: IPv4 → 4 bytes (e.g., 192.168.1.1)
- Assigned via:
    - DHCP (automatically)
        - DHCP (Dynamic Host Configuration Protocol) is a network protocol that automatically assigns IP addresses and other network configuration data to devices on a network
        - **Automatic IP Address Assignment**: Instead of manually configuring each device with a static IP address, DHCP automatically assigns a temporary IP address from a pool of available addresses.
        - **Network Configuration**: DHCP also provides other network configuration information, such as the **subnet mask, default gateway, and DNS server addresses**.
        - **Dynamic IP Addresses**: DHCP assigns IP addresses dynamically, meaning that the same IP address can be assigned to different devices at different times.
        - **Simplified Network Administration**: DHCP significantly simplifies network administration by automating the IP address assignment

process, reducing the need for manual configuration and potential errors.

- **Lease Management**: DHCP uses a lease system, where IP addresses are assigned for a limited time (the lease period). After the lease expires, the IP address can be reassigned to another device.
- Common Usage: DHCP is widely used in home networks, businesses, and other organizations where devices connect to a network.

- Static configuration (manually)

Network and Host Portion

- An IP address is made up of two parts:
  - **Network Part**: Identifies which network the device belongs to.
  - **Host Part**: Identifies the specific device (host) within that network.
- Example: IP address 192.168.1.10 with subnet mask 255.255.255.0
  - IP Address (in binary):
    - 192.168.1.10 → 11000000.10101000.00000001.00001010
  - Subnet Mask (in binary):
    - 255.255.255.0 → 11111111.11111111.11111111.00000000
  - Apply the Subnet Mask:
    - The **1s** in the mask indicate the **Network Part**
    - The **0s** in the mask indicate the **Host Part**

Portion	Binary	Meaning
Network	11000000.10101000.00000001	192.168.1 (Network)
Host	00001010	.10 (Specific device)

  - So, in this case:
    - Network part = 192.168.1.0
    - Host part = 10 (this is the specific computer)
- Why is this important?
  - It helps routers:
    - Send data to the correct network

        - Then, once on the network, deliver it to the correct host
    - Analogy:
        - Think of it like a mailing address:
            - Network Part = Street name
            - Host Part = House number

Subnet Mask

- A **subnet** (short for **subnetwork**) is a smaller, logical section of a larger network.
- Think of it as dividing a big group into smaller, manageable teams — each with its own members but still part of the main organization.

Why Subnet?

- Subnetting helps:
    - Helps determine if another device is in the **same subnet**.
    - Improve security (isolate traffic)
    - Improve performance (reduce congestion)
    - Efficiently use IP addresses
    - Organize large networks into logical groups (like departments)

Structure of an IP Address:

- Every IP address has:

IP Address = Network part + Host part

A **subnet mask** tells us **how many bits** are for the network and **how many** are for the host.

- Example:
    - IP: 192.168.1.10
    - Subnet Mask: 255.255.255.0
      (Binary: 11111111.11111111.11111111.00000000)
    - **Network Part**: First 24 bits → 192.168.1
    - **Host Part**: Last 8 bits → 10 (this device)
- This subnet has:
    - Network Address: 192.168.1.0
    - Broadcast Address: 192.168.1.255

- Usable Hosts: 254 (from .1 to .254)

- Example:
  - Your IP: 192.168.1.123
  - Destination: 192.168.1.2
  - Subnet mask: 255.255.255.0 (→ /24)
    - Same subnet? Yes → Direct MAC-based delivery.
    - Different subnet? Route via default gateway.

Subnet Mask	CIDR	# of Hosts	Usage Example
255.0.0.0	/8	~16 million	Very large networks
255.255.0.0	/16	~65,000	Large organizations
255.255.255.0	/24	254	Common office/home
255.255.255.192	/26	62	Small subnets

What is CIDR?

- CIDR stands for Classless Inter-Domain Routing.
- It's a modern way of representing IP addresses and their subnet masks, replacing the older "classful" system (Class A, B, C, etc.).
- CIDR writes an IP address like this: 192.168.1.0/24
- Where:
  - 192.168.1.0 is the network address
  - /24 means the first 24 bits are the network part
  - The remaining bits (32 – 24 = 8 bits) are for the host part

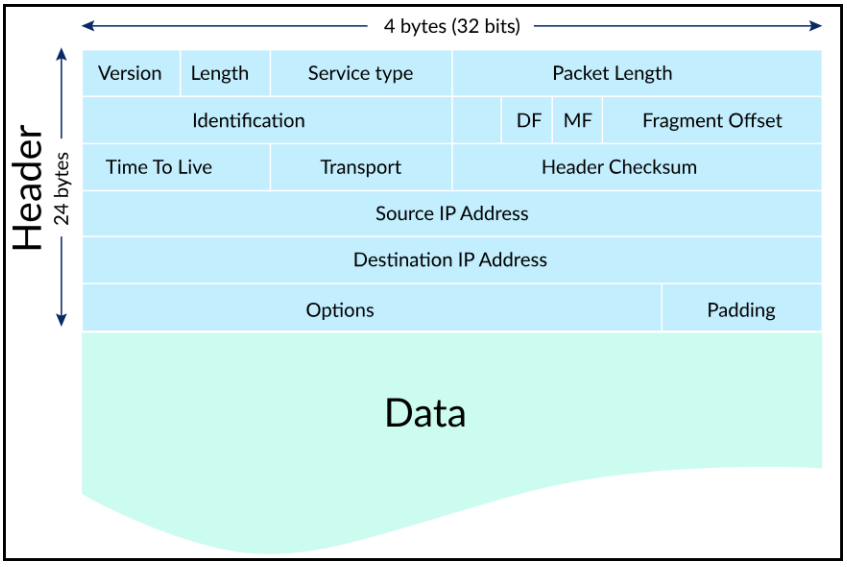
3. Default Gateway

- Acts as the door to the outside network.
- Crucial for routing traffic across networks.
- Hosts use the gateway when the destination is outside their subnet.
- Routers typically have multiple interfaces (IP addresses in multiple networks).
  - Routers connect multiple networks — they route data between them.

- To do this, they must be part of each network they connect to.
- That's why routers have:
  - Multiple network interfaces (like ports)
  - Each interface has its own IP address
  - Each IP belongs to a different network
- Example: Imagine a router connecting two networks:

Interface	**IP Address**	**Connected Network**
Interface 1	192.168.1.1	Network A: 192.168.1.0/24
Interface 2	10.0.0.1	Network B: 10.0.0.0/24

4. Anatomy of the IP Packet



Field	Description
Version	IPv4 or IPv6
Header Length	Usually 5 (×4 = 20 bytes)
Total Length	Size of entire packet (header + data)
Identification / Flags / Offset	Used for fragmentation
Time To Live (TTL)	Prevents infinite loops in routing
Protocol	Tells what's inside (TCP, UDP, ICMP, etc.)
Source IP	Where the packet came from
Destination IP	Where the packet is going

5. Fragmentation

- IP packets that exceed the MTU (Max Transmission Unit) need to be fragmented.
- MTU is typically 1500 bytes (Ethernet).
- Fragmentation splits a large packet into smaller chunks.
- Issues:
    - Reordering required
    - Harder to manage
    - Security risks (spoofed fragments)
- Most modern protocols avoid it (Don't Fragment flag).
- So, to clarify:
    - **Application layer** (e.g., a browser or chat app) just sends data — no knowledge of packets or MTU.
    - IP layer (Network Layer) does the fragmentation if needed, based on the MTU of the next hop.
    - TCP (Transport Layer) may also adjust segment size to avoid fragmentation altogether.
- Example:
    - You open a large file in an FTP client.
    - The FTP (application) sends a stream of data.
    - TCP breaks it into manageable chunks (segments).
    - IP tries to fit each segment into a packet of ≤ MTU (e.g., 1500 bytes).
    - If a packet is still too big, IP fragments it before sending it to the data link layer.

6. Time To Live (TTL)

- A counter that decreases at each router hop.
- If TTL = 0 → packet is discarded and ICMP message is sent back.
- Prevents infinite loops.
- Used by traceroute to map the path between source and destination.

What is ICMP?

- ICMP stands for Internet Control Message Protocol.

- It is a network layer protocol used by devices (like routers, switches, and computers) to send error messages and diagnostic information — **not to send data itself**.

What does ICMP do?

- ICMP helps:
  - Diagnose network issues
  - Report errors when IP packets can't reach their destination
  - Inform about things like **unreachable hosts** or networks, packet loss, **TTL expiry**, etc.
- Features:
  - No port required
  - Critical for debugging (ping, traceroute)
  - Often blocked by firewalls (may break some network tools)

Common Uses of ICMP

Tool / Scenario	**Description**
ping	Sends ICMP Echo Request → receives Echo Reply to check connectivity
traceroute / tracert	Uses ICMP (and/or UDP) to trace path through routers
Destination Unreachable	Router says, "I can't deliver this packet"
Time Exceeded (TTL)	"Packet expired before reaching its destination"
Fragmentation Needed	Can't send because MTU is too small

7. Protocol Field

- Tells what kind of payload is inside the packet:
  - 1: ICMP
  - 6: TCP
  - 17: UDP
- Helps routers and firewalls make decisions without reading the full packet.

- Types of Firewalls and Their OSI Layers:

Firewall Type	OSI Layer	What It Does
Packet-Filtering Firewall	**Layer 3 & 4** (Network & Transport)	Filters traffic based on **IP address**, **port number**, and **protocol**.
Stateful Firewall	**Layer 3 & 4**	Remembers connections and filters packets in the context of active sessions.
Application Firewall	**Layer 7** (Application)	Inspects **application-layer data** like HTTP requests, DNS queries, etc.
Next-Gen Firewall (NGFW)	**Layers 3–7**	Combines packet filtering, deep packet inspection, intrusion prevention, etc.
Web Application Firewall (WAF)	**Layer 7**	Specifically protects **web applications** from threats like SQL injection, XSS.

8. Ping & Traceroute

- Ping:
  - Sends ICMP Echo Request to a target.
  - Measures round-trip time.
- Traceroute:
  - Sends packets with increasing TTL (1, 2, 3, ...)
  - Each router along the path returns an ICMP "Time Exceeded"
  - Helps map all routers between you and a destination

IV. UDP

- User Datagram Protocol is a **Layer 4 transport protocol.**
- It's called **"User Datagram Protocol"** because:
  - It's a **transport protocol usable by applications** ("user processes")
  - It transmits **individual user-defined messages (datagrams)**.
  - Each message is treated independently — there's **no stream or connection**.
- In contrast:
  - Some other protocols (like **ICMP** or **IGMP** (Internet Group Management Protocol)) are meant for network control or system-level functions, not for end-user applications.
- UDP is stateless and connectionless.
  - No handshakes.
  - No connection tracking.
- You simply send a datagram — no guarantee it will arrive.

1. Characteristics of UDP

Feature	**Description**
Connectionless	No setup between client and server.
Stateless	No memory/state retained on server or client.
No Acknowledgment	You don't know if a message arrived.
No Retransmission	Lost packets are not resent.
Message-Oriented	Each UDP datagram is a complete message.
Small Header	8 bytes only, compared to TCP's 20+ bytes.
Fast & Lightweight	No sequencing, no ordering, no flow control.

2. Use Cases of UDP

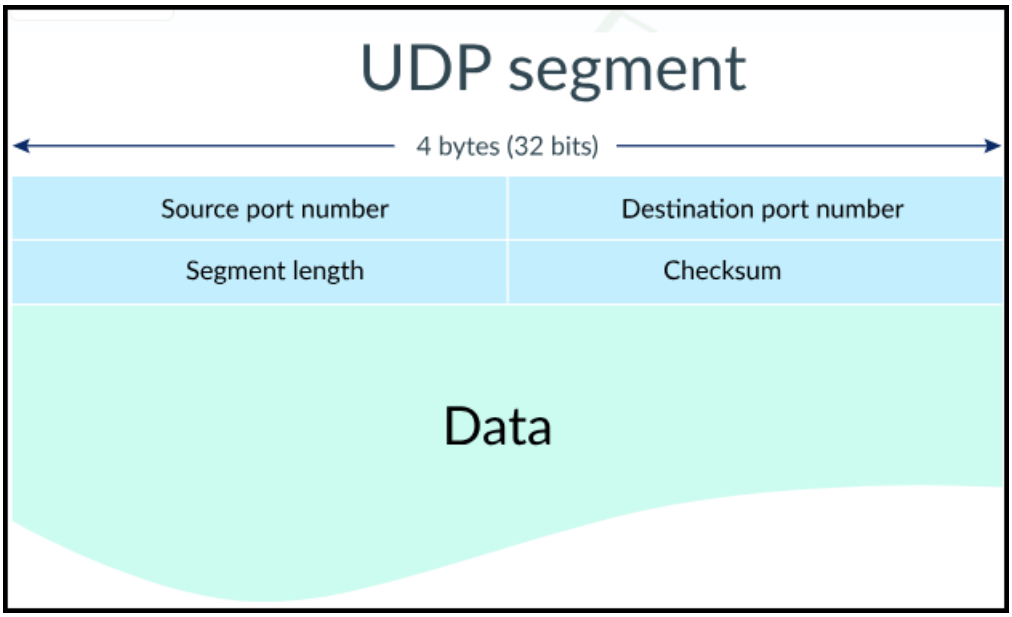
- 🎥 Video/Audio Streaming (loss-tolerant).
- 🎮 Gaming/Multiplayer (needs speed over reliability).
- 🔐 VPN Tunnelling (e.g., OpenVPN).
- 🌐 DNS Queries (fast one-shot queries to resolve hostnames).

- 🗣️ WebRTC (peer-to-peer browser communication).
- 👷 Custom Protocols (you build reliability if needed).

3. UDP Internals: Ports and Multiplexing

- Ports
  - Each process/application listens on a port (0–65535).
  - Ports allow UDP to address specific applications on the same machine.
- Multiplexing & Demultiplexing
  - Multiple apps (different ports) share one IP address.
  - UDP enables:
    - Multiplexing: Multiple apps → single network
    - Demultiplexing: Receiver identifies target app using destination port

4. UDP Packet Structure (Datagram Anatomy)



Fields:

- **Source Port (16 bits)**: App that sent the datagram.
  - Total ports possible $2^{16} - 1 = 65535$.
- **Destination Port (16 bits)**: App to receive the datagram.

- **Segment Length (16 bits)**: Size of **UDP header + data**. Again, the maximum length of a UDP segment is 65535 bytes.
- **Checksum (16 bits)**: Data integrity check (optional in IPv4)

5. Pros and Cons of UDP

- Pros
    - Low latency — no handshake or retransmission
    - Stateless — scalable; low memory usage on servers
    - Simple header — only 8 bytes
    - Broadcast/multicast support
    - Custom reliability possible at app level
- Cons
    - No delivery guarantee
    - No order or duplicate detection
    - No flow/congestion control
    - Easy to spoof (UDP flood/DNS amplification attacks)
    - No session awareness — harder to track client state

6. Security Considerations

- UDP is vulnerable to:
    - Spoofing (no handshake means anyone can send)
    - DDoS (Distributed Denial-of-Service: flooding with fake UDP requests)
        - It's a type of cyberattack where a malicious actor floods a targeted server or network with traffic, overwhelming it and making it unavailable to legitimate users
    - DNS Poisoning (modifying UDP DNS responses)
        - Domain Name System (DNS) poisoning happens when fake information is entered into the cache of a domain name server, resulting in DNS queries producing an incorrect reply, sending users to the wrong website. DNS poisoning also goes by the terms “DNS spoofing” and “DNS cache poisoning.”

- Protection Tips:
    - Rate-limit UDP responses
    - Validate source IPs (when possible)
    - Prefer TCP where reliability matters

V. TCP

- TCP stands for Transmission Control Protocol. It is a **Layer 4 protocol** (Transport Layer) in the OSI model.
- It is the **vehicle of choice** for client-server communication when reliable data delivery is needed.
- Compared to UDP (User Datagram Protocol), TCP **controls transmission**, ensuring data arrives completely and in order.

1. Key Features of TCP

- Reliable Transmission
    - Guarantees that all data sent will arrive correctly.
    - Lost packets are retransmitted.
- Ordered Delivery
    - Packets are received in the same order they were sent.
- Connection-Oriented
    - Requires a 3-way handshake before sending data.
- Stateful
    - Keeps connection info (IP + Port pairs) in memory via file descriptors.
- Flow Control
    - Ensures the sender doesn't overwhelm the receiver.
    - Example
        - Receiver's buffer can hold 16 KB.
        - It currently has 4 KB of data in it.
        - It advertises a window size of 12 KB.
        - The sender can send up to 12 KB of data before waiting for more ACKs.

- **Zero Window Condition**
  - If the receiver's buffer is full:
    - It sets the window size to 0.
    - This tells the sender: "Stop! I can't take any more right now."
  - The sender then:
    - Stops sending.
    - Periodically sends a "**Window Probe**" to check if the window size has increased.

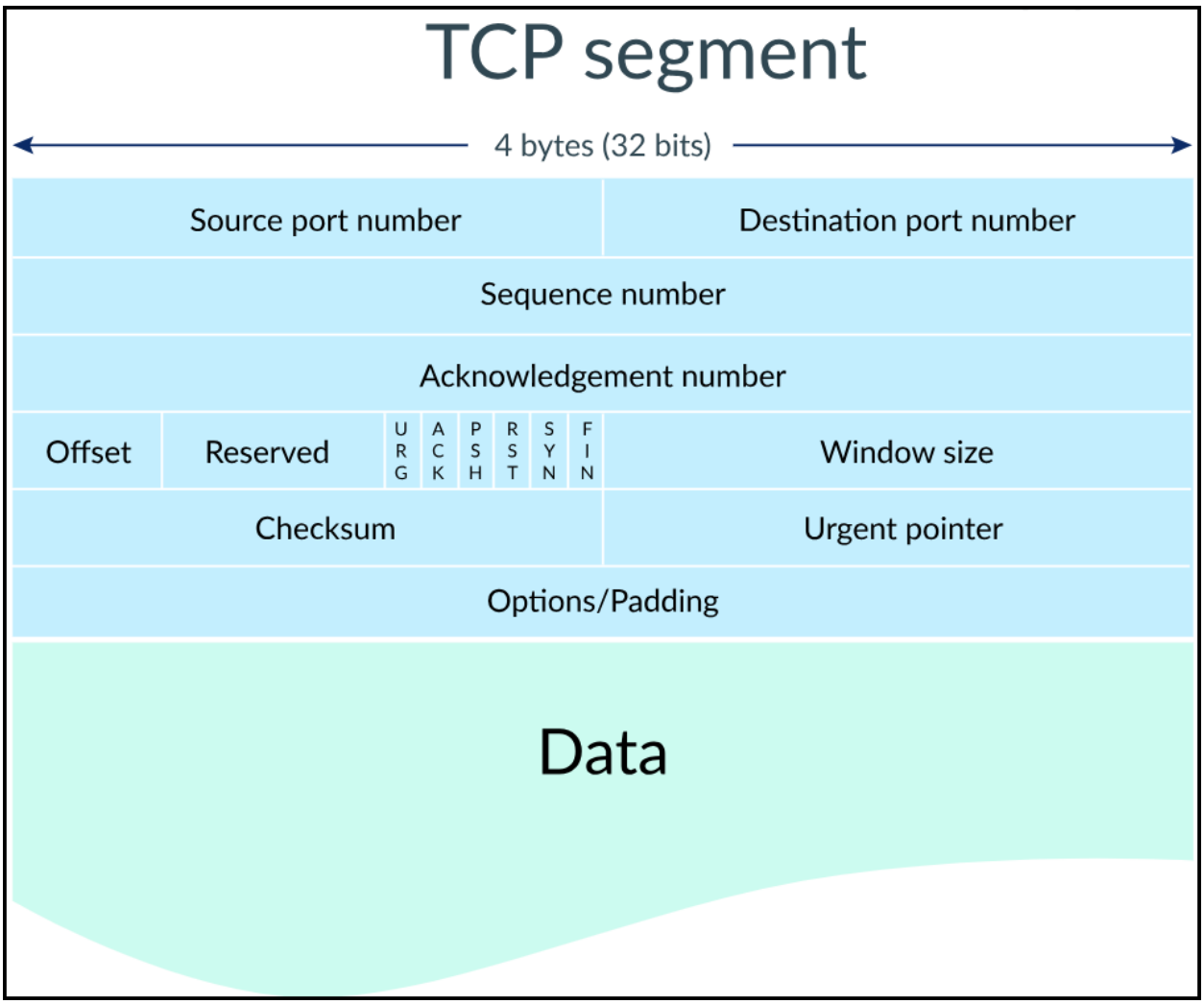
- Congestion Control
  - Slows down transmission to avoid network congestion.

2. TCP vs. UDP

Feature	TCP	UDP
Reliability	Yes (Guaranteed delivery)	No
Connection Required	Yes	No
Ordered Delivery	Yes	No
Flow/Congestion Ctrl	Yes	No
Header Size	20-60 bytes	8 bytes

3. TCP Segment Structure

- TCP Segment = TCP Header (20-60 bytes) + Data
- Key fields in header:
  - Source & Destination Ports (16 bits each)
  - Sequence Number: Track data ordering
  - Acknowledgment Number: Confirm received data
  - Flags: SYN, ACK, FIN, RST, PSH, URG, etc.
  - Window Size: Flow control
  - Checksum: Data integrity



Flags Overview

Flag	Purpose
SYN	Start connection
ACK	Acknowledge received data
FIN	Terminate connection
RST	Reset connection
PSH	Push data
URG	Urgent data
ECN/CWR	Congestion control (Explicit)

4. Connection Establishment (3-Way Handshake)

1. **Client sends SYN** (with sequence number X)
2. **Server replies with SYN+ACK** (with its sequence Y and ACK X+1)
3. **Client replies with ACK Y+1**

- Both client and server now track the connection using a **4-tuple**: (Source IP, Source Port, Destination IP, Destination Port)
- Stored in a structure called **file descriptor** (socket).

5. Data Transmission

- Each TCP packet is called a segment.
- Segments are sent with **incremental sequence** numbers to keep track of order.
- Receiver sends back **ACK** (acknowledgement) numbers:
    - E.g., if receiver got segment 1, 2 → **ACK 3** will be sent. (expects next segment)
- Segments may arrive out of order. TCP buffers and reorders them.
- What happens if a segment is lost?
    - Receiver won't send ACK for the lost segment.
    - Sender waits (timeout) and retransmits the lost segment.

6. Memory and Resource Usage

- TCP is **stateful**: both server and client store connection state (sequence number, window size, etc.)
    - OS maintains a connection table (hashed from IP + Port pairs).
    - Every new connection use:
        - Source IP, Source Port
        - Destination IP, Destination Port
        - Stored in memory (RAM), so there's a limit on active TCP connections.
    - Multiple apps can run on the same host using different ports.
    - OS hashes (IP + Port tuples) to map incoming segments to the correct application.

7. Connection Termination (4-Way Handshake)

- FIN → Sender wants to close.
- ACK → Receiver acknowledges FIN.
- FIN → Receiver also wants to close.
- ACK → Sender acknowledges.
- The initiator remains in a TIME_WAIT state to ensure no duplicate packets arrive.

8. Maximum Segment Size (MSS)

- Determined by MTU (Max Transmission Unit) of network:
  - Calculated as: $MSS = MTU - IP\ header - TCP\ header$
- Default MTU: 1500 bytes (common on Ethernet)
  - IP header: 20 bytes, TCP header: 20 bytes
  - So, MSS ≈ 1460 bytes

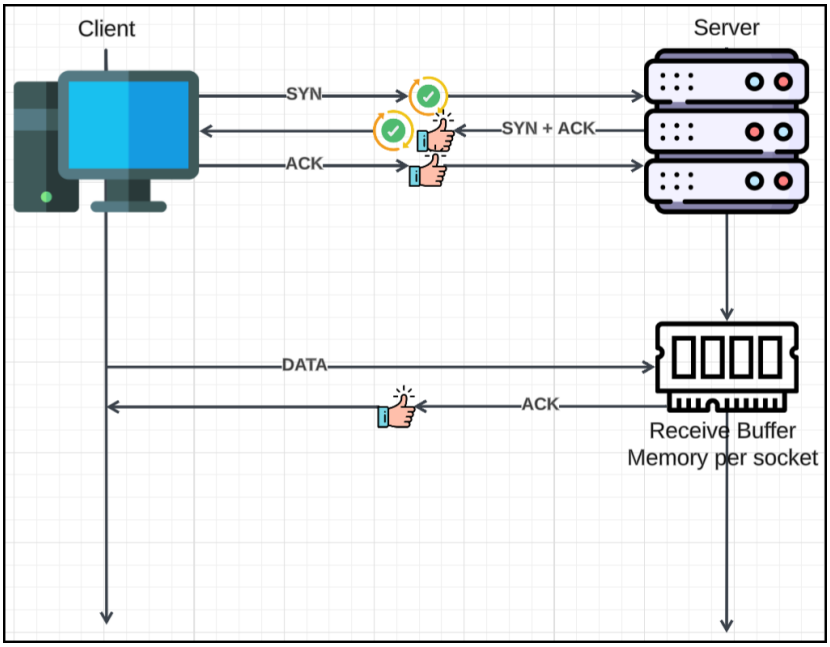
9. TCP Under Load: Connection Queue vs Memory Buffers

- When a TCP server is hit with **many incoming connections or large volumes of data**, two critical components play a role in managing that load:
  - Connection Queue (Before Connection Established)
  - Receive Buffer (After Connection Established)

Connection Queue (Before Connection Established)

- It's a **queue** in the **TCP stack** of the server that holds incoming connection requests.
- When a client sends a **SYN**, the server puts it in the **queue until the 3-way handshake is complete**.
- When a client wants to connect to a server, it performs the 3-way handshake:
  - Client → Server: Sends SYN (synchronize)
  - Server → Client: Replies with SYN-ACK
  - Client → Server: Replies with ACK
  - Only after step 3, the connection is considered established.
- TCP Connection Queue: Before Handshake is Complete

    - When the server receives a SYN (step 1), it:
        - Allocates memory (state) for that connection
        - Places it in a half-open connection **queue**, waiting for step 3 (ACK)
- Queue Overflow Risk:
    - The server sets a backlog limit (e.g., 128 connections).
    - If more clients send SYNs than the queue can hold, new connection attempts are **dropped or refused**.
- The Problem: SYN Flood Attack
    - A malicious client (attacker) sends thousands of SYNs, but never replies with ACK.
    - Result:
        - Server fills up the TCP half-open queue
        - Legitimate clients get dropped or delayed
        - Memory is consumed for incomplete handshakes
        - This is called a SYN Flood attack.
    - This is common during connection spikes or **DDoS attacks**.
- Optimization:
    - Increase backlog size using `listen(sock, backlog)`
    - Use SYN cookies to prevent resource exhaustion
        - A SYN cookie is a clever technique where the server:
            - Does not allocate memory for the connection after receiving SYN
            - Instead, it encodes the connection state into the Initial Sequence Number (ISN) in the SYN-ACK
            - When client responds with ACK, the server decodes the cookie and only then allocates memory
        - This way:
            - No memory is wasted on fake SYNs
            - The server remains responsive under attack



Receive Buffer (After Connection Established)

- Once a connection is **established**, each TCP socket gets a **receive buffer** in **kernel memory**.
- Incoming data (segments) from the sender are stored here **temporarily**, until the **application reads them**.
- Buffer Overflow Risk:
    - If the **application reads too slowly**, this buffer may fill up.
    - If full, TCP can't store new data — so it:
        - **Advertises a zero-window size** (flow control)
        - Tells the sender to **pause sending**
- Optimization:
    - Make sure the application reads fast (asynchronous or multi-threaded)
    - Tune buffer size via socket options (SO_RCVBUF)
    - Avoid long blocking operations on critical threads

VI. Transport Layer Security

- TLS (Transport Layer Security) is a standard encryption protocol.
- It protects communication between two parties (e.g., client and server).
- Without TLS, data is sent in plain text, which can be read by anyone in the middle (like ISPs or routers).

Where TLS Fits in the OSI Model

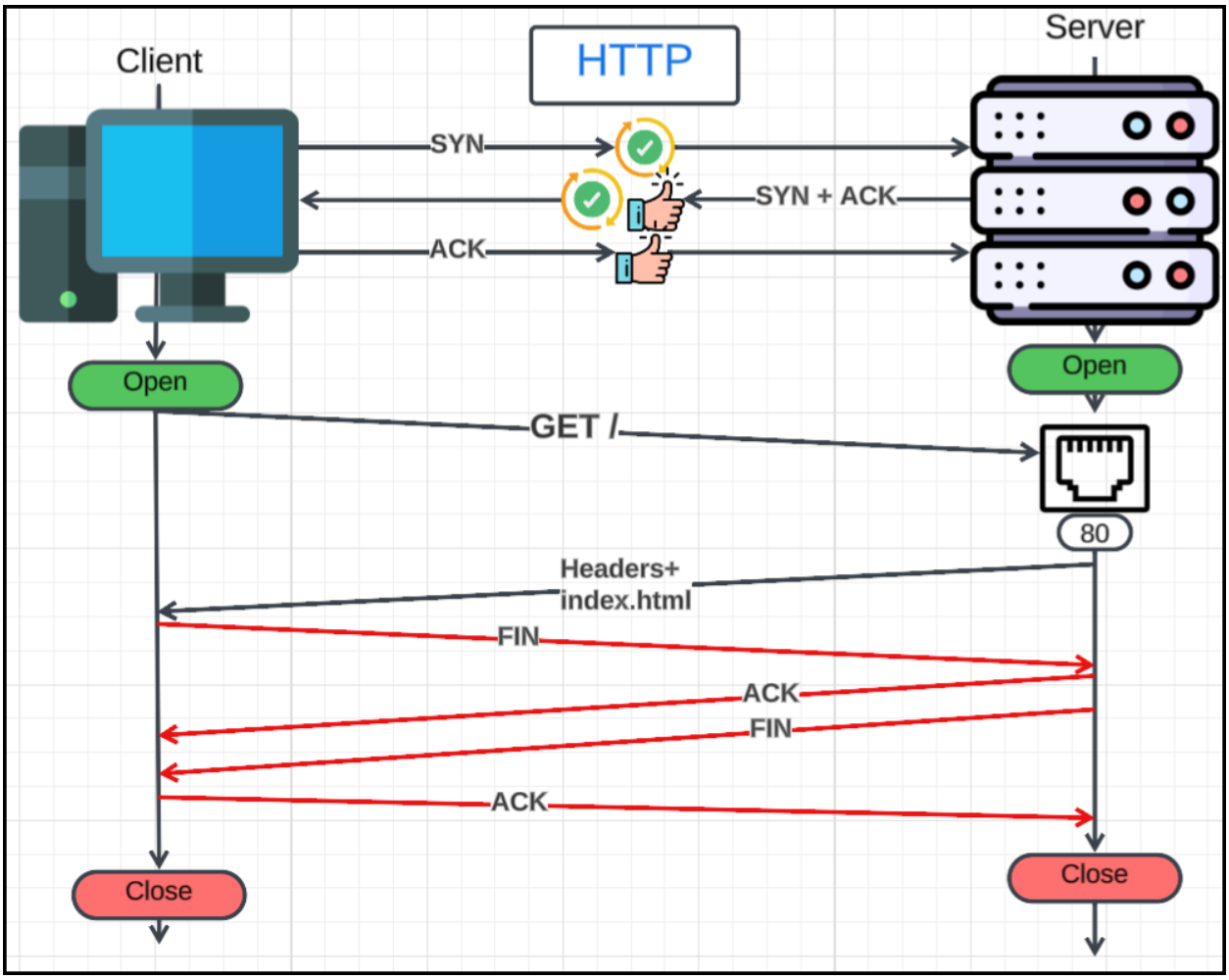
- TLS isn't tied to a single OSI layer, but if you had to pick, it most closely fits in Layer 5 (Session Layer):
  - Because it maintains session state.
  - It sits above TCP (Layer 4) and below HTTP (Layer 7).

1. HTTP over TCP

- To establish a connection before sending any data, TCP performs a **3-step handshake**:

Step	Direction	Description
1 **SYN**	Client → Server	Client sends a **SYN** (synchronize) packet to initiate connection.
2 **SYN+ACK**	Server → Client	Server replies with **SYN + ACK**, acknowledging the request.
3 **ACK**	Client → Server	Client sends final **ACK** to complete the handshake.

- After this handshake, both client and server have an open TCP connection.
- TCP ensures reliable, ordered, and error-checked delivery of data.
- Once TCP connection is open, the application layer (HTTP protocol) begins:
  - Client sends an `HTTP GET` request: `GET /`
  - Requests the root document (e.g., `index.html`).
  - Uses `port 80` (default for HTTP).
  - No encryption (this is plain HTTP, not HTTPS).
- Server processes the GET request. Sends back:
  - `HTTP Headers` (`status code`, `content type`, etc.)
  - Body: the contents of the requested file (e.g., `index.html`)
- This response is split into TCP segments if large.



Connection Termination

Once the request and response are complete:

- Client closes the TCP connection.
- This can also involve a TCP 4-step teardown (not shown in detail here):
    - `FIN` from client
    - `ACK` from server
    - `FIN` from server
    - `ACK` from client

Port Details

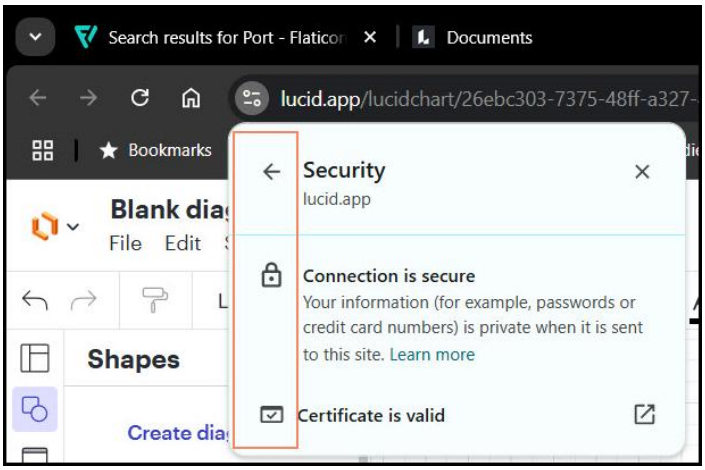
- Client uses a random high-numbered port (ephemeral port).
- Server listens on port 80 (standard for HTTP).
- Ports help in multiplexing many connections simultaneously.

Summary Table

Layer	Protocol	Action
4	TCP	Establish connection with 3-way handshake
7	HTTP	Send GET request from client to server
4	TCP	Receive HTTP response in segments
4	TCP	Connection termination (FIN/ACK)

2. HTTPS Communication (HTTP + TLS + TCP)

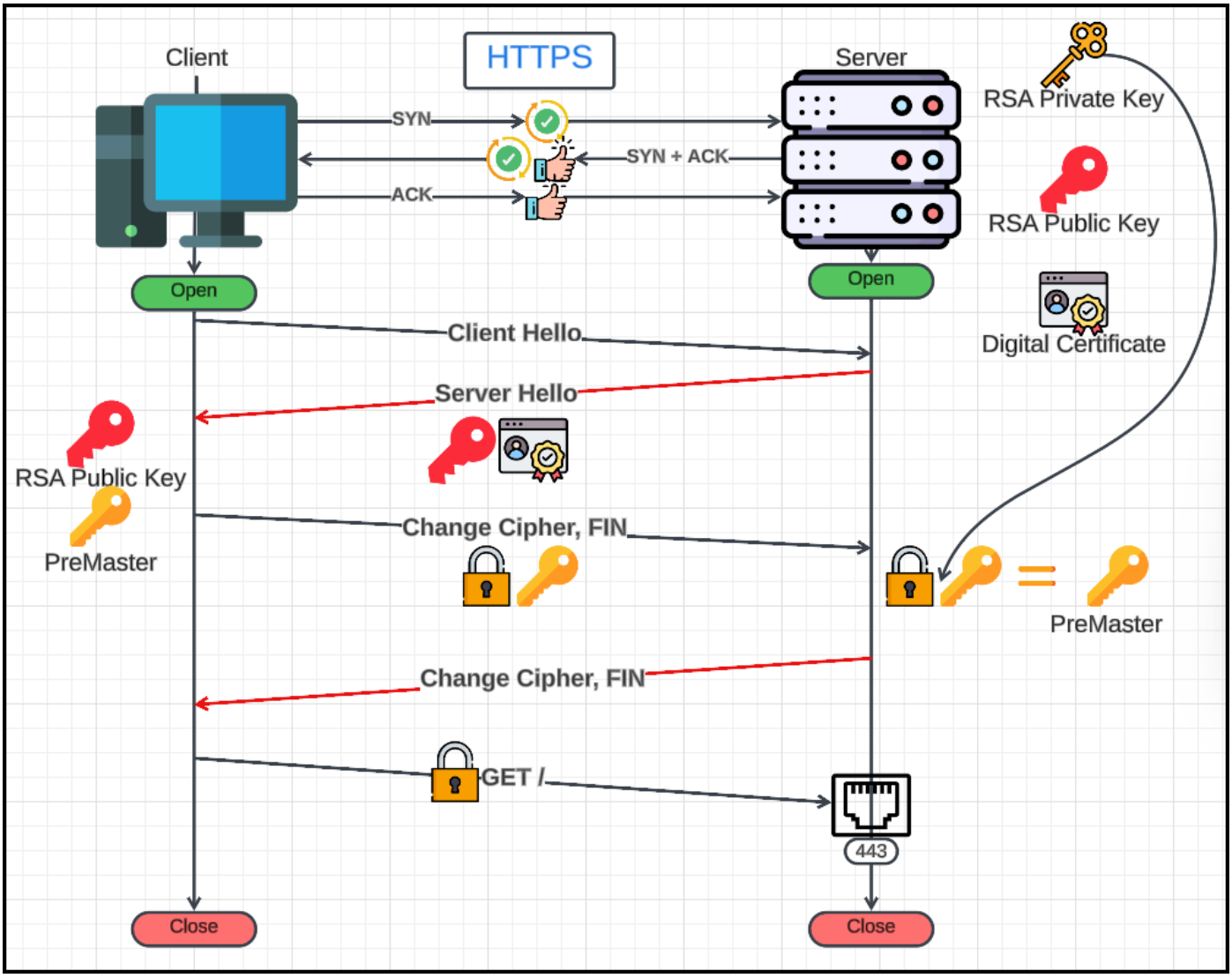
- HTTPS = **HTTP** over **TLS** over **TCP**
- It ensures that HTTP traffic is:
  - **Encrypted**
  - **Authenticated**
  - **Tamper-proof**
- Default Port: **443**
- Used by browsers to securely access websites (🔒 in the address bar)



A. TCP 3-Way Handshake (Same as HTTP)

- Purpose: **Establish a reliable connection**.
- Sequence:
  1. Client → Server: SYN
  2. Server → Client: SYN + ACK
  3. Client → Server: ACK
- Now a reliable TCP channel is open.

B. TLS Handshake (Before HTTP Starts)



Phase 1: Negotiation & Authentication

1. ClientHello (sent by client)

- Proposes:
    1. TLS version
    2. Supported cipher suites (algorithms for encryption/hash/key exchange)
    3. Supported compression methods
    4. A random number: ClientRandom
    5. Optional extensions like SNI (Server Name Indication)

2. ServerHello (sent by server)

- Chooses:
    1. TLS version
    2. Cipher suite from client's list
    3. Compression method
    4. Sends its own random number: ServerRandom
    5. Sends its digital certificate containing its public key
    6. May include session ID or extensions

Phase 2: Key Exchange

3. Client Key Exchange

- Client generates a **pre-master** secret (randomly).
- Encrypts the pre-master secret using server's public key 🔑 (from the certificate).
- Sends the encrypted pre-master secret to the server.

4. Server decrypts

- Server uses its private key 🔑 to decrypt the received encrypted pre-master secret.
- Now both client and server have the same **pre-master secret**.

Why Pre-Master Secret?

- It is not directly used for encryption.
- It's combined with **ClientRandom** and **ServerRandom** to derive the master secret.

5. Master Secret Derivation

- Using a pseudo-random function (PRF):

```
MasterSecret = PRF(pre_master_secret, "master secret", ClientRandom + ServerRandom)
```

- This **Master Secret** is used to generate:
    - Client MAC key
    - Server MAC key
        - **MAC = Message Authentication Code**
            - It's a cryptographic checksum.
            - It ensures data integrity (detects tampering) and authenticity (who sent it).

- Think of it like a digital fingerprint attached to each encrypted message.

- **How are they used?**
  - When a message is sent (e.g., encrypted application data):
    i. Sender encrypts the message using the symmetric encryption key (like AES).
    ii. Sender calculates a MAC over the plaintext + sequence number.
    iii. Appends the MAC to the message.
    iv. Sends the encrypted message + MAC.
  - When a message is received:
    i. Receiver decrypts the message.
    ii. Recalculates the MAC over the decrypted content.
    iii. Compares with received MAC:
      a. If it matches → ✅ message is authentic and intact.
      b. If it doesn't → ❌ tampering or transmission error.

- Client encryption key
- Server encryption key
- Initialization vectors (IVs), if needed

6. ChangeCipherSpec

- Both sides send this message to inform each other:

  "*From now on, I will use the negotiated symmetric encryption and MAC keys.*"

7. Finished Messages

- Each side sends a Finished message:
  - Encrypted using the newly established symmetric encryption key.
  - Contains a hash of all handshake messages seen so far, to verify handshake integrity.

Now a secure encrypted tunnel is ready.

C. HTTP Request (Encrypted Inside TLS)

- Client sends:

```
GET / HTTP/1.1
Host: example.com
User-Agent: ...
```

- But this is now **encrypted using TLS**, so middlemen cannot read it.

D. HTTP Response (Encrypted Too)

- Server sends:
  - Encrypted HTTP headers (e.g., `200 OK`)
  - Encrypted body (e.g., `HTML`, `images`, etc.)

E. Connection Termination

- Either side may initiate TLS session closure (sending encrypted `close_notify`).
- Then follows a TCP connection teardown (`FIN + ACK` exchange).

3. Why Not Use RSA for Everything?

- RSA is slow for large data.
- It's used only to securely exchange the symmetric key.

4. Certificates and Authentication

- The server proves its identity using a certificate.
- Certificate includes a public key and is signed by a Certificate Authority (CA).
- Clients check that the CA is trusted.
- This process is built on PKI (Public Key Infrastructure).

VII. HTTP/1.1

- HTTP (HyperText Transfer Protocol) is a protocol used to exchange information between a client (like browsers or apps) and a server.
- Common implementations: **Clients** include browsers, Python/JavaScript apps; **Servers** include Nginx, Apache, Tomcat, Tornado, etc.
- Versions:
    - HTTP 1.0 (Old, simple, but inefficient)
    - HTTP 1.1 (Most widely used, improvements over 1.0)
    - HTTP/2 (Faster, multiplexing, binary protocol)
    - HTTP/3 (Uses QUIC, built on UDP for better performance)
    - Why the slash and a single digit?
        - The reason for the change to /2 and /3 instead of, say, 2.0 or 3.0, is to emphasize that these are not just incremental updates like 1.1 was to 1.0. They represent fundamental redesigns of how HTTP operates "on the wire." Signifies a **major architectural shift and a new binary framing layer** for the protocol.

1. Structure of an HTTP Request

An HTTP request has 5 main parts:

a. Method:

Method: Specifies the action the client wants to perform on the resource.

- **GET**: Used for reading or retrieving data. (Most common for web Browse).
- **POST**: Used for submitting data to be processed (e.g., submitting a form, uploading a file).
- **PUT**: Used for updating or creating a resource at a specific URL.
- **DELETE**: Used for removing a resource.
- **HEAD**: Similar to **GET**, but only requests the headers (no body).
- Note for Web Development: In practical web development, GET and POST are by far the most frequently used.

b. Path

This is the specific part of the URL *after* the domain name.

- **Example**: For www.google.com/about, the path is /about.
- If you visit just www.example.com, the path is /.
- Query Parameters: These are appended to the path to send additional data with GET requests.
    - Format: ?key1=value1&key2=value2
    - Important: Query parameters are part of the URL.
- URL Length Limit: URLs have a length limit (defined by the server, often 2000-8000 characters/bytes). Be cautious with very long URLs, especially for GET requests, as they might fail on some servers.

c. Protocol

Specifies the HTTP version being used.

- Examples: HTTP/1.0, HTTP/1.1, HTTP/2, HTTP/3.

d. Headers (Key-Value Pairs)

Provide **metadata** about the **request** or the **client**.

- **Host Header**: Crucial and now required. Specifies the domain name of the server.
    - Why it's important: A single IP address can host multiple websites (this is called "multi-homed hosting" or "virtual hosting"). The Host header tells the server which website on that IP address the client wants to access. Without it, the server wouldn't know which site to serve.
- **Content-Type**: Describes the type of content in the request body (e.g., **application/json**, **text/html**).
- **Content-Length**: Indicates the size of the request body in bytes.
- **Cookie**: Contains cookies sent by the client to the server for session management.
- **User-Agent**: Identifies the client software making the request (e.g., browser name and version).

Example HTTP Request

```
curl -v google.com
> GET / HTTP/1.1
> Host: google.com
> User-Agent: curl/8.13.0
> Accept: */*
```

e. Body (Optional)

The actual data being sent with the request.

- **GET** requests typically do not have a body. Data is sent via query parameters in the URL.
- **POST** requests typically have a body. This is where data like form submissions, JSON payloads, or uploaded files are sent.
- Body Size Limit: The body generally does not have a strict size limit defined by the protocol itself, but rather by the Content-Length header and server configuration.

2. Structure of an HTTP Response

An HTTP response from the server also has a specific structure:

a. Protocol

- The HTTP version used by the server in its response (e.g., HTTP/1.1).

b. Status Code

A three-digit number indicating the outcome of the request.

- **200 OK**: Request successful.
- **301 Moved Permanently**: Resource has permanently moved to a new URL.
- **302 Found**: Resource has temporarily moved.
- **304 Not Modified**: The client's cached version of the resource is still valid.
- **404 Not Found**: The requested resource could not be found.
- **500 Internal Server Error**: A generic error on the server side.

c. Status Text

A human-readable description of the status code (e.g., OK, Not Found).

- In HTTP 1.1, this text is redundant (as 200 always means OK), and it was removed in HTTP/2 and HTTP/3. This change caused some minor issues with browser developer tools that relied on this text for displaying descriptions.

d. Headers

Provide metadata about the response or the server.

- **Location**: Used in 3xx redirects to specify the new URL.
- **Date**: The date and time the response was generated.
- **Content-Type**: Describes the type of content in the response body (e.g., **text/html**, **application/json**).
- **Content-Length**: Indicates the size of the response body in bytes.
- **Server**: Identifies the web server software (e.g., Nginx, Apache).

Example:

```
< HTTP/1.1 301 Moved Permanently
< Location: http://www.google.com/
< Content-Type: text/html; charset=UTF-8
< Date: Mon, 16 Jun 2025 12:29:21 GMT
< Expires: Wed, 16 Jul 2025 12:29:21 GMT
< Cache-Control: public, max-age=2592000
< Server: gws
< Content-Length: 219
< X-XSS-Protection: 0
< X-Frame-Options: SAMEORIGIN
```

e. Body

The actual content of the response (e.g., HTML page, JSON data, image file).

Example:

```
<html>
  <head>
    <meta http-equiv="content-type" content="text/html;charset=utf-8" />
    <title>301 Moved</title>
  </head>
  <body>
    <h1>301 Moved</h1>
    The document has moved
    <a href="http://www.google.com/">here</a>.
  </body>
</html>
```

3. HTTP 1.0: The Original Approach

- **Connection Model:** "Request-Response-Close" model.
    - For **every single request**, a new TCP connection was opened, the request was sent, the response was received, and then the connection was immediately closed.
    - Steps**:**
        1. Open TCP connection (3-way handshake).
        2. Send HTTP Request.
        3. Receive HTTP Response.
        4. Close TCP connection.
- **Drawbacks:**
    - **High Latency:** Opening a new TCP connection for every resource is very expensive due to the overhead of the TCP 3-way handshake.
    - **Resource Inefficiency:** Thrashed servers with numerous connection setups and tear-downs.

- **No Buffering/Chunking:** The entire response had to be sent at once. You couldn't send parts of a response incrementally.
    - This meant no "Transfer-Encoding: chunked" (important for long-polling, server-sent events, or responses where the length is unknown beforehand).
- **Optional Host Header:** Made multi-homed websites (hosting multiple domains on one IP) difficult or impossible to implement reliably.

4. HTTP 1.1: Improvements and Persistent Connections

HTTP 1.1 addressed many of HTTP 1.0's limitations, becoming the backbone of the web for a long time.

- **Persistent Connections ("Keep-Alive"):** This is the most significant improvement.
    - Clients can send a Connection: keep-alive header to request that the server keep the TCP connection open after sending a response.
    - This allows multiple requests and responses to be exchanged over a ***single* TCP connection**, significantly reducing overhead.
    - **Benefits:**
        - **Lower Latency:** No need to re-establish a TCP connection for every resource.
        - **Lower CPU Usage:** Fewer connection setups and tear-downs on both client and server.
        - More efficient use of network resources.
- **Streaming with Transfer-Encoding: chunked:**
    - This header allows the server to send the response body in multiple "chunks" of data, without knowing the total Content-Length beforehand.
    - **Benefits:** Enables long-polling, server-sent events, and streaming large files where the size isn't immediately known.
- **Required Host Header:** Officially mandated the Host header, fully enabling multi-homed hosting.

- **Pipelining (Rarely Used/Disabled by Default):**
    - **Concept:** Allows the client to send multiple requests over a single connection *without waiting for the response to the previous request*.
    - **Analogy:** Like a CPU pipeline where different stages of multiple instructions are processed concurrently, or doing laundry (washing one load, drying another, folding a third simultaneously).
    - **Problem (Head-of-Line Blocking):** Even if a later, smaller resource's response is ready first, the server *must* send responses in the order the requests were received. If the first requested resource is very large, it blocks all subsequent responses, even if they are ready sooner.
    - **Further Problem:** Proxies in the middle of the network might not maintain the request order, leading to unpredictable results.
    - **Conclusion:** Due to these issues, HTTP 1.1 pipelining is **almost always disabled by default** in browsers and servers. It's largely considered a failed feature.
- **HTTP Smuggling (Security Vulnerability):**
    - A critical issue related to the parsing of HTTP 1.1 requests, especially when proxies or load balancers are involved.
    - Occurs when different servers (e.g., a frontend Nginx proxy and a backend Node.js server) interpret HTTP headers like Content-Length or Transfer-Encoding differently.
    - Attackers can craft requests that appear valid to one server but are interpreted differently by another, potentially "smuggling" malicious requests that bypass security controls (e.g., accessing admin pages). This highlights the complexity of HTTP parsing and the importance of consistent implementations.

5. HTTP/2 Overview

- Originally developed by Google as SPDY.
- Uses binary protocol (not plain text). Unlike text-based HTTP 1.x, HTTP/2 uses a binary protocol. This makes it more efficient to parse and reduces errors.
- Key features:
    - **Header and body compression**.
    - **Multiplexing**: multiple streams on one connection. Allows multiple requests and responses to be sent concurrently over a *single TCP connection*. Each request/response is assigned a unique "stream ID," so order doesn't matter, eliminating HTTP 1.1's head-of-line blocking problem.
    - **Stream IDs help track request-response pairs**.
    - **Secure** by default due to protocol ossification (middleboxes assume HTTP/1 format).
        - Why HTTP/2 Needs HTTPS?
            - Protocol Ossification:
                - Old network devices expect HTTP 1.1 syntax → They might break HTTP/2 traffic.
                - Encryption prevents middleboxes from interfering.
    - **Server Push** (now deprecated): proactively sends resources to client. A feature where the server could proactively send resources to the client that it anticipated the client would need (e.g., CSS and JavaScript files for an HTML page).

6. HTTP/3 (Over QUIC)

- **Fundamental Change:** Replaces TCP as the underlying transport protocol with **QUIC (Quick UDP Internet Connections)**.
    - **Why QUIC?** The main limitation of HTTP/2 is still TCP's inherent "**head-of-line blocking**" at the transport layer. If a single TCP packet is lost, it can block all other streams over that TCP connection until it's retransmitted.

7. Summary

Feature	HTTP/1.0	HTTP/1.1	HTTP/2	HTTP/3
Connection	Closed	Persistent	Persistent	Persistent
Multiplexing	❌	❌ (limited)	✅ (via streams)	✅ (via QUIC)
Pipelining	❌	✅ (broken)	✅ (with stream IDs)	✅
Binary Format	❌	❌	✅	✅
TLS Required	❌	❌	✅ (usually)	✅ (always)
Host Header	Optional	Required	Required	Required

VIII. HTTPS, TLS, and Encryption

1. HTTP and HTTPS Fundamentals

- **HTTP (Hypertext Transfer Protocol)** is a request-response system protocol built on top of TCP, utilizing TCP as a streaming protocol for massive message-based communication.
- **HTTPS (Hypertext Transfer Protocol Secure)** is the secure version of HTTP. The 'S' indicates that the communication is encrypted.
    - HTTPS = HTTP + **TLS encryption** (previously SSL).
    - Example: https://example.com means communication is encrypted up to the first server (like Cloudflare), but not necessarily end-to-end.
- **Encryption Necessity**: Encryption is crucial to prevent intermediaries, such as Internet Service Providers (ISPs), from snooping on requests and responses.
- **TLS (Transport Layer Security)**, formerly known as SSL (Secure Sockets Layer), is the protocol added to HTTP to achieve HTTPS.
    - **Works at Layer 6/7 (Application Layer)** in the OSI model.
    - Encrypts data before passing it to TCP (Layer 4).
    - TLS can be thought of as an application protocol that handles encryption just before the application layer.
    - The kernel (OS) simply sends bytes over IP and port; it doesn't care if the data is encrypted or not. The encryption happens at a higher layer before data reaches the kernel.
- **End-to-End Encryption Misconception**: It's important to understand that HTTPS encryption isn't necessarily end-to-end. Encryption occurs between your client and the first destination (e.g., a CDN or proxy). If a CDN or API gateway is involved, they might decrypt the content to cache or process it, then re-encrypt it before sending it to the actual backend.

2. Encryption Types

Two primary types of encryption:

Symmetric Encryption

- **Mechanism**: Uses a single key for both encrypting and decrypting content.
- **Speed**: It is extremely fast because it uses simple CPU instructions.
- **Challenge**: The main difficulty lies in securely exchanging this shared key between two parties.
- **Example**: AES (Advanced Encryption Standard) is a popular symmetric encryption algorithm, often working with block sizes like 128 bytes.

Asymmetric Encryption

- **Mechanism**: Employs two distinct keys: a public key and a private key.
    - **Public Key**: Can be shared with anyone.
    - **Private Key**: Must be kept secret.
- **Key Relationship**: You can generate the public key from the private key, but it's computationally impossible to derive the private key from the public key.
- **Encryption/Decryption**:
    - Content encrypted with the private key can only be decrypted by the corresponding public key.
    - Content encrypted with the public key can only be decrypted by the corresponding private key.
- **Speed**: Asymmetric encryption is computationally intensive and slow, making it unsuitable for encrypting large amounts of data.
- **Algorithms**: RSA is a common algorithm supporting asymmetric encryption. Key sizes (e.g., 256-bit, 2048-bit, 4096-bit) need to increase over time as computational power (like quantum CPUs) improves to break weaker keys.
- **Applications**:
    - **Proving Authenticity**: Encrypting with a public key and decrypting with a private key proves the authenticity of the recipient. Only the private key owner can decrypt messages encrypted with their public key.

- **Digital Signatures**: Encrypting with the private key and decrypting (verifying) with the public key is used for digital signatures.
    - This allows the sender to prove they created a document (by signing it with their private key).
    - Anyone with the sender's public key can verify the signature, ensuring the document hasn't been tampered with. If the document is altered, the signature will no longer match the content.

3. Certificates

- **Purpose**: Certificates are used to prove authenticity when connecting to a server, ensuring you are connecting to the legitimate server you intend to.
- **Structure**: A certificate is essentially metadata that carries a public key, along with other useful information like:
    - Version
    - Signature algorithm
    - Digital signature
    - Issuer's name (who issued the certificate)
    - Subject's name (owner of the certificate)
    - Subject Alternative Name (the website domain itself)
- **Generation**:
    - Generate a pair of private and public keys.
    - Place the public key and the website name into the certificate.
    - Sign the certificate using the private key (the private key itself is not included in the certificate).
- **Standard**: X.509 is the current standard for building certificates.
- **Self-Signed Certificates**: These are certificates where the private key used to sign the certificate is the same as the public key contained within it. They are often not trusted in public environments but might be used for local server setups.

4. Certificate Authorities (CAs)

For public internet use, certificates are signed by trusted Certificate Authorities.

- CAs themselves have certificates signed by other CAs, forming a "chain of trust."
- This chain eventually leads to a **Root Certificate**, which is a self-signed certificate but is universally trusted.
- Root certificates are pre-installed on computers and operating systems in a "certificate store." Browsers may also have their own certificate stores.

5. Certificate Verification:

- When a client receives a certificate (often just the "leaf" certificate), it checks who issued it (the CA).
- The client then looks up that CA in its certificate store.
    - The **certificate store** is a secure storage location on your device or within your browser where trusted **root certificates** and other **public key infrastructure (PKI) certificates** are kept. These stores are used by your operating system and browsers to verify the authenticity of servers during HTTPS connections.
- If found, it checks who signed the CA's certificate, tracing the chain back to a trusted root certificate.
- If a trusted root certificate is found, the certificate and the connection are considered trustworthy.
    - Sometimes, servers send the "full chain" of certificates because clients might not have all intermediate CAs in their store.
    - "Man-in-the-middle" attacks can occur if a malicious entity issues a fake certificate signed by a root certificate that a user's system implicitly trusts (e.g., from a compromised OS installation).

6. TLS Handshake (Simplified)

- **Key Exchange**: TLS uses symmetric keys (session keys) for actual data encryption because it's faster for large amounts of data.
- **Initial Exchange**: Since parties don't initially know each other's symmetric keys, asymmetric encryption is used to securely exchange these session keys.
- **Client Hello**: The client initiates the TLS handshake with a "Client Hello" message after the TCP handshake.
- **Server Response**: The server responds with its certificate, which contains its public key. This certificate is crucial for authentication, preventing a hacker from simply sending their own public key.

IX. WebSockets

It is a powerful protocol that enables real-time, bidirectional communication between a client (like a web browser) and a server.

1. Why WebSockets? The Limitations of HTTP

- **TCP is Bidirectional:** At its core, TCP (Transmission Control Protocol) is a bidirectional protocol, meaning data can flow in both directions simultaneously over a single connection.
- **Web Browsers and TCP:** Directly exposing raw TCP connections to web browsers would be a security nightmare. A malicious webpage could potentially access any TCP service (like an SMTP server) running on a user's machine or network, making it highly dangerous.
- **HTTP's Request-Response Model:** HTTP (especially HTTP 1.0 and 1.1) is fundamentally a request-response protocol.
    - **HTTP 1.0:** Opens a connection, sends a request, gets a response, and *closes* the connection. Very inefficient for multiple interactions.
    - **HTTP 1.1:** Introduced "persistent connections" (Keep-Alive), allowing multiple requests and responses over a *single, long-lived TCP connection*. This was a huge improvement.
- **The Need for "Push" Communication:** Even with persistent connections, HTTP 1.1 still operates on a request-response model. If a server needs to send data to a client *without* the client first requesting it (e.g., a new chat message, a live stock update), the client has to repeatedly "poll" the server (send requests at intervals to check for new data), which is inefficient and introduces latency.

2. How WebSockets Work

WebSockets leverage HTTP 1.1's persistent connections to "upgrade" to a new protocol.

- **HTTP as a Bootstrap:** WebSockets are built *on top* of HTTP. They start as a regular HTTP request.

- **The Handshake:**
    1. **Client Request:** The client sends a standard HTTP GET request, but includes special headers:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

        - `Upgrade: websocket`: Tells the server the client wants to switch to the WebSocket protocol.
        - `Connection`: Upgrade: A "hop-by-hop" header indicating that this is an upgrade request.
        - `Sec-WebSocket-Key`: A unique, randomly generated key used by the server to prove it understands the WebSocket protocol.
        - `Sec-WebSocket-Protocol`: (Optional) Specifies desired sub-protocols (e.g., chat), allowing the backend to understand the application-level context.
    2. **Server Response:** If the server supports WebSockets, it responds with:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGnim5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

        - `HTTP/1.1 101 Switching Protocols`: This status code indicates that the server is agreeing to switch protocols.
        - `Upgrade: websocket`
        - `Connection: Upgrade`

        - `Sec-WebSocket-Accept`: A hash generated by combining the client's Sec-WebSocket-Key with a specific GUID and then SHA1-hashing and base64-encoding the result. This confirms the server understands the handshake.
        - `Sec-WebSocket-Protocol`: (If sub-protocol was negotiated) The agreed-upon sub-protocol.
- **Bidirectional Communication:** After a successful handshake, the underlying TCP connection is no longer used for HTTP requests. Instead, it becomes a pure, raw, **bidirectional (full-duplex)** channel for WebSocket messages.
    - Both client and server can send messages to each other at any time, without waiting for a request or response.
    - This eliminates the need for polling and enables true real-time communication.
- **Protocol Scheme:** WebSockets use their own URL schemes:
    - `ws://` for unencrypted WebSocket connections.
    - `wss://` for secure WebSocket connections (over TLS/SSL, just like HTTPS).

3. Use Cases for WebSockets

WebSockets are ideal for applications requiring real-time updates and low-latency communication:

- **Chat Applications:** Instant messaging (e.g., WhatsApp Web, Discord, Twitch chat).
- **Live Feeds:** Stock tickers, sports scores, news updates.
- **Multiplayer Gaming:** Real-time game state synchronization.
- **Collaborative Tools:** Shared document editing, whiteboards.
- **Progress Indicators/Live Logs:** Streaming logs or progress updates from a server to a client.
- **IoT (Internet of Things):** Real-time device communication.

X. HTTP/2

1. HTTP/1.1 Refresher

- How HTTP/1.1 Works
    - Single Request per Connection:
        - Each HTTP/1.1 request blocks the TCP connection until the response is received.
        - Example: Loading HTML → CSS → JS → Images sequentially.
    - Workaround: Connection Pooling
        - Browsers open 6 parallel TCP connections (per domain) to load resources concurrently.
        - Why 6? Arbitrary limit chosen by Chrome for balance between performance and resource usage.
- Limitations of HTTP/1.1
    - Head-of-Line (HOL) Blocking:
        - Slow responses (e.g., a large image) delay subsequent requests on the same connection.
    - No Header Compression:
        - Headers (like Cookie, User-Agent) are sent in plaintext for every request.
    - Inefficient Resource Loading:
        - Requires multiple TCP handshakes (high latency for new connections).

2. HTTP/2 Introduction

- HTTP/2 is the modern version of HTTP that improves performance over HTTP/1.1.
- It's designed for speed and efficiency, using one TCP connection for multiple requests.
- Replaces many limitations of HTTP/1.1 like connection bottlenecks and redundant headers.
- HTTP/2 introduces several key concepts that fundamentally change how communication occurs.

3. Multiplexing over a Single Connection:

- **Core Idea:** Instead of multiple TCP connections, HTTP/2 uses a *single* TCP connection for all communication with a server.
- **Stream IDs:** Every request and response is tagged with a unique "stream ID." Clients typically use odd-numbered IDs (e.g., 1, 3, 5), and servers use even-numbered IDs for streams they initiate (though a server responding to a client's stream would use the client's stream ID for that response).
- **Concurrency:** This allows the client to send all requests concurrently on that single TCP connection.
- **Out-of-Order Processing:** The server can process these streams concurrently (if it's multi-threaded) and send back responses *out of order*. For example, a small image requested later can be sent back before a large HTML file requested earlier. The stream ID ensures the client correctly associates the response with its original request.

4. Header Compression:

- HTTP/2 compresses request and response headers using HPACK algorithm. This significantly reduces the amount of data transferred, especially beneficial for applications with many small requests.
- (In HTTP 1.1, header compression was disabled due to security vulnerabilities like the CRIME attack.)

5. Server Push (Deprecated/Replaced):

- **Original Idea:** Allowed the server to proactively "push" resources to the client that it anticipated the client would need (e.g., sending CSS and JavaScript files along with an HTML page without the client explicitly requesting them).
- **Why it Failed:**
    - **Lack of Client Knowledge:** The server doesn't know what's already cached on the client, leading to wasted bandwidth by pushing already-available resources.

    - **Client Confusion:** The web model traditionally involves one request leading to one response. Server push broke this, as clients would suddenly receive multiple responses without initiating multiple requests, making client-side implementation complex.
    - **Complexity:** Configuring and managing server push effectively became very difficult.
- **Replacement:** The concept has largely been superseded by "Early Hints" (HTTP 103 status code), which is a hint from the server to the client about resources it might need, allowing the client to proactively fetch them.

6. Secure by Default (Practical Enforcement):

- While HTTP/2 technically can run over unencrypted connections, in practice, all major browsers only support HTTP/2 over TLS (HTTPS).
- **Reason (Protocol Ossification):** Older network middleboxes (routers, firewalls) were designed to understand HTTP 1.1's text-based format. When HTTP/2's binary format came along, these middleboxes would often reject it if it was unencrypted, as they didn't recognize the traffic. Encrypting the traffic makes it opaque to these middleboxes, allowing it to pass through undisturbed.
- **Negotiation:** The use of HTTP/2 is negotiated during the TLS handshake via the Application-Layer Protocol Negotiation (ALPN) extension.

7. Pros and Cons of HTTP/2

Pros:

- **Multiplexing:** The biggest advantage. Achieves true concurrency over a single connection, eliminating head-of-line blocking inherent in HTTP 1.1 and reducing connection overhead.
- **Resource Efficiency:** Saves resources on both client and server by requiring fewer TCP connections.
- **Header and Data Compression:** Reduces bandwidth usage.
- **Secure by Default:** Encourages and practically enforces the use of HTTPS, enhancing security for web traffic.

Cons:

- **TCP Head-of-Line Blocking (Remaining Issue):** This is HTTP/2's biggest limitation. While HTTP/2 solves *application-layer* head-of-line blocking (where one slow HTTP stream blocks others), it doesn't solve *TCP-layer* head-of-line blocking.
    - **How it Works:** TCP ensures ordered delivery of segments (chunks of data). If *any* TCP segment for *any* stream on that single TCP connection is lost or delayed, all subsequent segments on that *same TCP connection* must wait for the lost segment to be retransmitted and reordered, even if they belong to different, independent HTTP/2 streams. This can still cause delays for all streams on that connection.
- **Increased Server CPU Usage:**
    - HTTP/2's binary framing layer, stream management (stream IDs, flow control at the stream level), and header compression require more complex parsing and processing on the server side compared to the simpler text-based parsing of HTTP 1.1.
    - This additional work consumes more CPU cycles on the backend, which might require more server resources (e.g., more containers or Kubernetes pods) to handle the same load, especially under high traffic.
- **Server Push Failure:** As discussed, this feature didn't pan out and is no longer used, adding to the protocol's complexity without delivering intended benefits.
- **Workload Dependency for Performance:** HTTP/2 is not *always* faster. Its benefits are most pronounced when a client needs to fetch *many* resources concurrently from the *same domain* (e.g., a complex webpage with many images, CSS, and JS files).
    - If an application primarily sends single, infrequent requests or if requests are processed extremely quickly on the backend, the overhead of HTTP/2's parsing might negate its performance advantages.
    - Understanding your application's workload is key to deciding if HTTP/2 (or even HTTP/3) is truly beneficial.

XI. Many ways to HPPTS

We will explore the various **styles of HTTPS communication**, focusing on how different combinations of protocols and configurations affect **latency**, performance, and efficiency. Rather than just saying "use HTTPS," we dive into the layers and how they can be optimized:

1. Connection establishment
2. Encryption setup
3. Data transfer
4. Connection teardown (closing)

1. The Three Pillars of HTTPS Communication

Every HTTPS connection fundamentally involves three expensive steps that contribute to latency:

1. **Establishing a Connection (TCP Handshake):**

- This is the initial setup phase where the client and server establish a TCP connection.
- It involves a "three-way handshake" (SYN, SYN-ACK, ACK) to ensure both parties are ready to communicate.
- This step incurs network round trips (RTT - Round Trip Time) and introduces latency.

2. **Establishing TLS (TLS Handshake):**

- After the TCP connection is established, the client and server perform the TLS handshake.
- This is where they:
    - Exchange certificates (for authentication and trust verification).
    - Negotiate cryptographic algorithms.
    - Securely exchange symmetric "session keys" for encrypting the actual data.
- This step involves additional round trips and is computationally intensive due to asymmetric encryption.

3. **Sending Data:**

- Once the TCP and TLS connections are established, and symmetric keys are agreed upon, the client and server can finally send and receive encrypted application data (HTTP requests/responses).

- The goal is to keep the connection open for as long as possible to send as much data as possible, avoiding the repeated overhead of steps 1 and 2.

2. Variants of HTTPS Communication

Each of the following is a different way to perform HTTPS, with trade-offs in **latency, complexity, and compatibility**:

- HTTPS over TCP with TLS 1.2
- HTTPS over TCP with TLS 1.3
- HTTPS over QUIC (Will not be discussed)
- HTTPS over TCP Fast Open (TFO)
- HTTPS over TCP with TLS 1.3 with 0-RTT
- HTTPS over QUIC with 0-RTT (Will not be discussed)

3. HTTPS: The Secure Way to Communicate

- **HTTPS (Hypertext Transfer Protocol Secure)** is the secure version of HTTP. It's what you see in your browser's address bar when you visit a secure website (e.g., https://google.com).
- **Running on TCP:** HTTPS typically runs "over TCP" (Transmission Control Protocol). Think of TCP as a reliable delivery service for your data.
- **Supported Protocols:** Only **HTTP/1.1** and **HTTP/2** work on top of TCP for HTTPS.
- **HTTP/3 and QUIC:** A newer version, **HTTP/3**, does *not* use TCP. Instead, it uses **QUIC (Quick UDP Internet Connections)**, which is a different transport protocol. This means HTTP/3 establishes connections differently.

TLS 1.2: The Minimum Standard for Security

- **TLS (Transport Layer Security)** is the cryptographic protocol that provides secure communication over a computer network. It's the "S" in HTTPS.
- **TLS 1.2 (Version 1.2)** is the minimum version discussed and used in modern secure connections.

- **Deprecated Versions:** Older versions like TLS 1.0 and 1.1 are considered insecure and are deprecated (meaning they are outdated and should not be used, though some systems might still support them for backward compatibility).

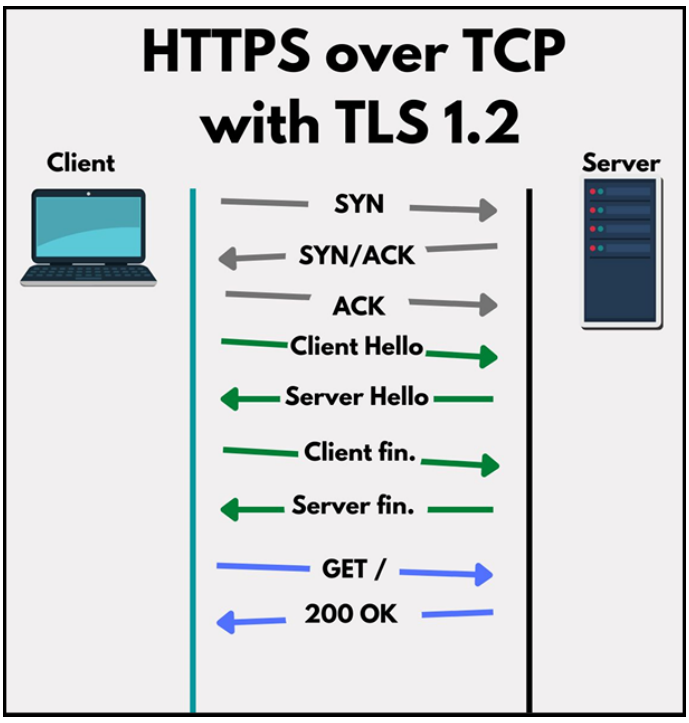
4. HTTPS Over TCP with TLS 1.2

Phase 1: The TCP Three-Way Handshake (Establishing the Foundation)

This is the very first step and creates a basic, *unencrypted* connection.

1. **SYN (Synchronize):** The client sends a "SYN" packet to the server, essentially saying, "Hello, I want to connect!"
2. **SYN-ACK (Synchronize-Acknowledge):** The server receives the SYN, acknowledges it, and sends back a "SYN-ACK" packet, saying, "Hello back! I received your request, and I'm ready to connect."
3. **ACK (Acknowledge):** The client receives the SYN-ACK and sends back an "ACK" packet, saying, "Great! I received your acknowledgment."

- **Result:** At this point, a **stateful TCP connection** is established. Both the client and server are aware of the connection and its state. However, any data sent now would be in **plaintext** (unencrypted) and visible to anyone "sniffing" the network. This is why we need the next step.



Phase 2: The TLS Handshake (Securing the Connection with Encryption)

This is the crucial part where the client and server agree on how to encrypt their communication. The goal is to agree on a **symmetric key**.

- **Symmetric Encryption:** This type of encryption uses the *same key* to both encrypt and decrypt data. Examples include AES (Advanced Encryption Standard) or ChaCha20. It's very fast once the key is established.
- **The Challenge:** How do the client and server secretly agree on this shared symmetric key without anyone in the middle being able to see it? They can't just send the key in plaintext! This is where **key exchange algorithms** come in.

Here's the detailed flow of the TLS Handshake:

1. **Client Hello:**
    - The client sends a "Client Hello" message to the server.
    - This message is a "buffet" of information, including:
        - Supported **symmetric key encryption algorithms** (e.g., AES 128, AES 256).
        - Supported **key exchange algorithms** (e.g., RSA, Diffie-Hellman, ECDH - Elliptic Curve Diffie-Hellman).
        - TLS extensions, such as SNI (Server Name Indication), which helps the server know which website the client is trying to access if multiple websites are hosted on the same IP address.
2. **Server Hello:**
    - The server receives the "Client Hello" and reviews the client's supported options.
    - The server then **negotiates** and chooses the strongest and most compatible encryption and key exchange algorithms from the client's list (e.g., "Let's use AES 256 for symmetric encryption and ECDH for key exchange").
    - The server sends back a "Server Hello" message, which includes:
        - Its chosen key exchange algorithm parameters (often a public key part).
        - Its chosen symmetric encryption algorithm.

        - Other server-side parameters.

3. **Key Exchange (Client's Turn):**
    - The client receives the "Server Hello" and the server's public parameters.
    - Using the agreed-upon key exchange algorithm, the client generates its own **private parameters** and combines them with the server's **public parameters**.
    - Crucially, this process allows the client to **independently generate the symmetric key**.
    - The client then sends its **public parameters** back to the server (e.g., "Client Key Exchange" message).
4. **Key Exchange (Server's Turn):**
    - The server receives the client's public parameters.
    - The server, using its own private parameters and the client's public parameters (and the agreed-upon key exchange algorithm), can now **independently generate the exact same symmetric key** that the client generated.
    - At this point, **both the client and server possess the identical symmetric key.**
5. **Change Cipher Spec & Finished Messages:**
    - **Client Change Cipher Spec & Finished:** The client sends a "Change Cipher Spec" message to indicate that all subsequent communication will be encrypted. It then sends a "Finished" message, which is an encrypted message that confirms the successful completion of the handshake and that the symmetric key is correctly established.
    - **Server Change Cipher Spec & Finished:** The server receives the client's "Finished" message, decrypts it (using the newly established symmetric key), and verifies it. If successful, the server sends its own "Change Cipher Spec" and an encrypted "Finished" message.

- **Result:** The TLS Handshake is complete! Both the client and server now have the same symmetric key, and all further communication will be encrypted using this key.

Data Transfer (Encrypted Communication)

- **Encrypted Requests:** Now, the client can send its actual request (e.g., an HTTP GET request for a web page) to the server. This request is **encrypted** using the symmetric key.
- **Server Decryption and Processing:** The server receives the encrypted data stream, uses the symmetric key to **decrypt** it, and then processes the original HTTP request (e.g., retrieves the requested web page).
- **Encrypted Responses:** The server then prepares its response (e.g., the 200 OK status code and the web page content), **encrypts** it with the symmetric key, and sends it back to the client.
- **Client Decryption:** The client receives the encrypted response and uses the symmetric key to **decrypt** it, displaying the web page to the user.

Summary: TLS 1.2 Workflow

Step	Description
1	Establish TCP connection (SYN → SYN-ACK → ACK)
2	Client sends “Client Hello” (supported ciphers, key exchange algorithms)
3	Server responds with “Server Hello” (chooses cipher & algorithm)
4	Key exchange (securely generate shared symmetric key)
5	Both sides now share symmetric key
6	Client sends encrypted request
7	Server decrypts, processes, and replies encrypted

5. HTTPS Over TCP with TLS 1.3

TLS 1.3 is the latest version of the TLS protocol used for securing data over HTTPS.

Compared to TLS 1.2, TLS 1.3 offers:

- Faster connection setup (fewer round-trips)
- Stronger security (removes outdated algorithms)
- Simplified protocol flow

Layers Involved

- HTTPS still runs on top of TCP.
- So, it still requires:
    - A TCP 3-way handshake (SYN → SYN-ACK → ACK).
- TLS 1.3 is used with:
    - HTTP/1.1 or HTTP/2 (because HTTP/3 uses QUIC, not TCP).

Phase 1: The TCP Three-Way Handshake (Foundation Remains)

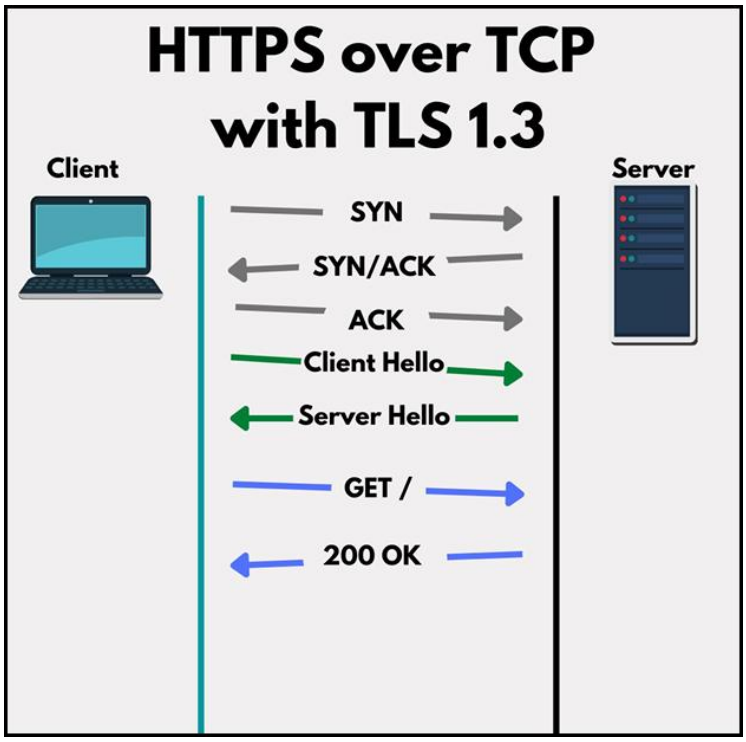
- This part is identical to TLS 1.2.
    1. **Client SYN:** Client sends a SYN to the server.
    2. **Server SYN-ACK:** Server responds with SYN-ACK.
    3. **Client ACK:** Client sends ACK.
- **Result:** A basic, unencrypted, stateful TCP connection is established between the client and server.

Phase 2: The TLS 1.3 Handshake (Optimized for Speed)

This is where TLS 1.3 shines by being more proactive and efficient.

1. **Client Hello (Proactive Key Exchange):**
    - Instead of sending a "buffet" of options and waiting for the server to pick, the client in TLS 1.3 makes an educated guess.
    - The client **assumes** a preferred **key exchange algorithm** (e.g., Elliptic Curve Diffie-Hellman - ECDH) and **immediately sends its public parameters** for this algorithm.

- To be safe, the client might even send public parameters for **multiple different key exchange algorithms** in case the server doesn't support the first choice.
- The "Client Hello" also still includes:
  - Supported **symmetric encryption algorithms** (e.g., AES 256, ChaCha20).
  - TLS extensions.



2. **Server Hello & Immediate Key Generation:**
   - The server receives the "Client Hello" and, if it agrees with one of the key exchange algorithms and public parameters sent by the client, it proceeds.
   - The server **generates its own private parameters** for the chosen key exchange algorithm.
   - **Crucially, the server immediately combines its private parameters with the client's public parameters to compute the symmetric key.** This means the server has the symmetric key *before* sending its response back to the client.
   - The server's response (which is still called a "Server Hello," but now combined with other messages) includes:

        - Its own **public parameters** for the key exchange (which the client needs to generate the symmetric key).
        - The **selected symmetric cipher suite** (e.g., AES 256).
        - The **server's digital certificate** (for authentication).
        - Crucially, this response also includes the **"Server Finished" message**, which is the first message encrypted with the newly generated symmetric key, confirming the handshake completion from the server's side.
3. **Client Key Generation & Certificate Verification:**
    - The client receives the server's response.
    - It uses the server's public parameters (received in the Server Hello) and its own private parameters to **compute the exact same symmetric key** that the server generated.
    - The client then **verifies the server's digital certificate** to ensure that it is communicating with the legitimate server and not an imposter. (This step is critical for trust and security and was also present in TLS 1.2, though less explicitly mentioned in the previous lecture summary.)
    - After successful key generation and certificate verification, the client sends its **"Client Finished" message**, which is also encrypted with the symmetric key.

- **Result:** Both client and server now possess the shared symmetric key, and the handshake is complete in just one round trip!

Encrypted Data

- Once the symmetric key is established and the certificate is verified, **encrypted application data can flow immediately.**
- The client sends encrypted requests, the server decrypts them, processes them, encrypts responses, and sends them back. All communication is secured by the shared symmetric key.

Quick Recap

Step	Description
1	TCP handshake (SYN → SYN-ACK → ACK)
2	ClientHello (key exchange + ciphers + extensions)
3	ServerHello (key exchange + certificate)
4	Both sides derive **same symmetric key**
5	Client verifies certificate
6	Encrypted data transfer begins

6. HTTPS over TCP Fast Open (TFO)

- What Is 0-RTT?
    - 0-RTT stands for "Zero Round Trip Time." It's an enhancement to TLS 1.3 that allows a client to send application data along with the very first message of the TLS handshake, effectively eliminating one full "round trip" of communication latency.
    - Goal: The primary goal of 0-RTT is to drastically reduce latency and make subsequent connections to a familiar server feel almost instant.
    - Still over TCP: Like regular TLS 1.3, 0-RTT still operates on top of TCP, meaning the initial TCP three-way handshake (SYN, SYN-ACK, ACK) is still a prerequisite.

The Prerequisite for 0-RTT

- 0-RTT is not for brand new connections. It relies on a previous secure connection having occurred between the specific client and server.
    - Pre-Shared Key (PSK): During a previous TLS 1.3 session, the client and server can establish and "share" a pre-shared key (PSK). This key is securely derived and stored by both parties for future use.
    - Session Resumption: 0-RTT is a form of session resumption, where a client can quickly resume a past secure session without going through the full key exchange process again.

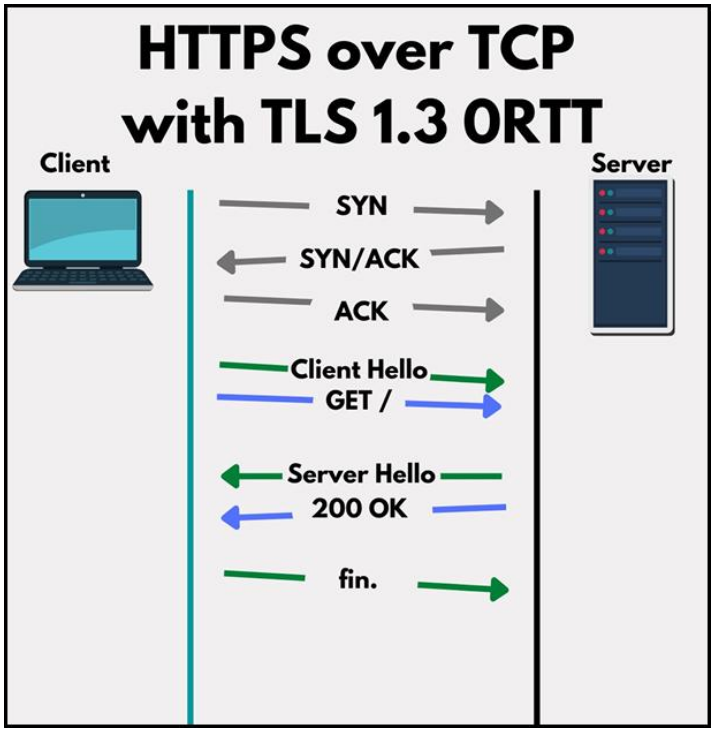
Phase 1: The TCP Three-Way Handshake (Standard)

- This initial setup remains the same:
    - Client SYN
    - Server SYN-ACK
    - Client ACK
- Result: A basic, unencrypted TCP connection is ready.

Phase 2: The TLS 1.3 0-RTT Handshake (Accelerated)

This is where the magic happens, leveraging the pre-shared key.

1. **Client Hello with Pre-Shared Key Hint and Encrypted Data:**
    - The client, having connected to this server before, sends a Client Hello message.
    - This Client Hello includes a **TLS extension** called pre_shared_key. This extension contains a "hint" (an identifier) about the pre-shared key that was established in a previous session.
    - **Crucially, *at the same time* (in the same packet), the client immediately encrypts its initial application data (e.g., an HTTP GET request) using the remembered pre-shared key and sends it along with the Client Hello**.



2. **Server Hello and Immediate Decryption/Processing:**
    - The server receives the Client Hello with the pre-shared key hint and the encrypted application data.
    - It looks up the pre-shared key associated with that hint.
    - If the pre-shared key is valid and accepted, the server can **immediately decrypt and process the client's application data** (the GET request).

- The server then sends back a Server Hello acknowledging the use of the pre-shared key. This Server Hello also includes its own Finished message (encrypted).

3. **Client Finished:**
   - The client receives the Server Hello and its Finished message, confirming the server accepted the 0-RTT request.
   - The client sends its own Finished message (also encrypted).

- **Result:** The client's application data has already been received and processed by the server **within the first "round trip"** of the TLS part of the connection. The server can even respond to the request very quickly.

7. TLS 1.2 RSA vs TLS 1.2 vs TLS 1.3.

- TLS Uses **Both** Asymmetric and Symmetric Encryption:

Type	Purpose
Asymmetric	For securely exchanging secrets (like a pre-master key) using public/private key.
Symmetric	For actual data encryption after the secret is shared. Much faster.

Protocol	Key Exchange Method	Uses RSA Encryption for Secret?	Asymmetric Used For
TLS 1.2 (RSA)	RSA	✅ Yes	Encryption + Cert verification
TLS 1.2 (DH/ECDHE)	(EC)DHE (Ephemeral)	❌ No	Key agreement + Cert verification
TLS 1.3	Always ECDHE (no RSA)	❌ No	Key agreement + Cert verification

XII. Backend Execution Patterns

1. How TCP Connections Are Established

- This section provides a crucial understanding of how network connections, particularly TCP connections, are established and managed, from the initial client request all the way to your backend application.

TCP Connection Establishment: The 3-Way Handshake

- TCP is stateful — Both client and server maintain state about the connection (like sequence numbers).
- 3 Steps in the TCP Handshake:
    - SYN: Client sends a "synchronize" request with its initial sequence number.
    - SYN-ACK: Server responds with its own sequence number and acknowledges the client's.
    - ACK: Client acknowledges the server's sequence number.
- After this, a connection is established.

Server Listening: Address and Port Explained

For a server application to accept connections, it must "listen" on a specific network address and port.

- **Address and Port:** You never just listen on a port; you *always* listen on an **IP address and a port number**.
    - **Port:** A logical number (e.g., 80 for HTTP, 443 for HTTPS, 8080 for many development servers) that identifies a specific application or service on a machine.
    - **IP Address:** Identifies the specific network interface on your computer.
- **Multiple Network Interfaces (NICs):** A single computer can have several network interfaces, each with its own IP address:
    - **Loopback Address:** 127.0.0.1 (IPv4) or ::1 (IPv6). This is for communication *within the same machine*.

- **Local Network Address:** E.g., 192.168.1.3 (private IP address assigned by your router).
- **Public IP Address:** An IP address accessible from the internet (e.g., 74.x.x.x if hosted on a cloud VM).

- **Bad Practice: Listening on All Interfaces (0.0.0.0):**
  - Many development frameworks (like Node.js) default to listening on 0.0.0.0 (or :: for IPv6), which means "listen on **all available network interfaces**."
  - **Why it's bad:** If you're building an internal API (e.g., an admin panel, a database API) meant only for local access or specific private networks, listening on 0.0.0.0 exposes it to the public internet if your machine has a public IP address.
  - **Security Risk:** This is a common reason why databases (like Elasticsearch, MongoDB) get accidentally exposed, leading to data leaks, especially if default or weak credentials are used.
  - **Best Practice:** Always **explicitly specify the IP address** you intend to listen on (e.g., 127.0.0.1 for local-only, a specific private IP for internal services) to control exposure. Databases should almost never be directly exposed to the public internet.

Interface	Type
`127.0.0.1`	Loopback (local)
`192.168.X.X`	Local network
`0.0.0.0`	All interfaces
`::1`	IPv6 loopback

The Kernel's Role in Connection Management

The operating system's **kernel** plays a critical role in managing network connections, separate from your application process.

- **Separation of Concerns:**
  - **Kernel Space:** The kernel runs in its own isolated memory space. It handles low-level networking, device drivers, and core OS functions.

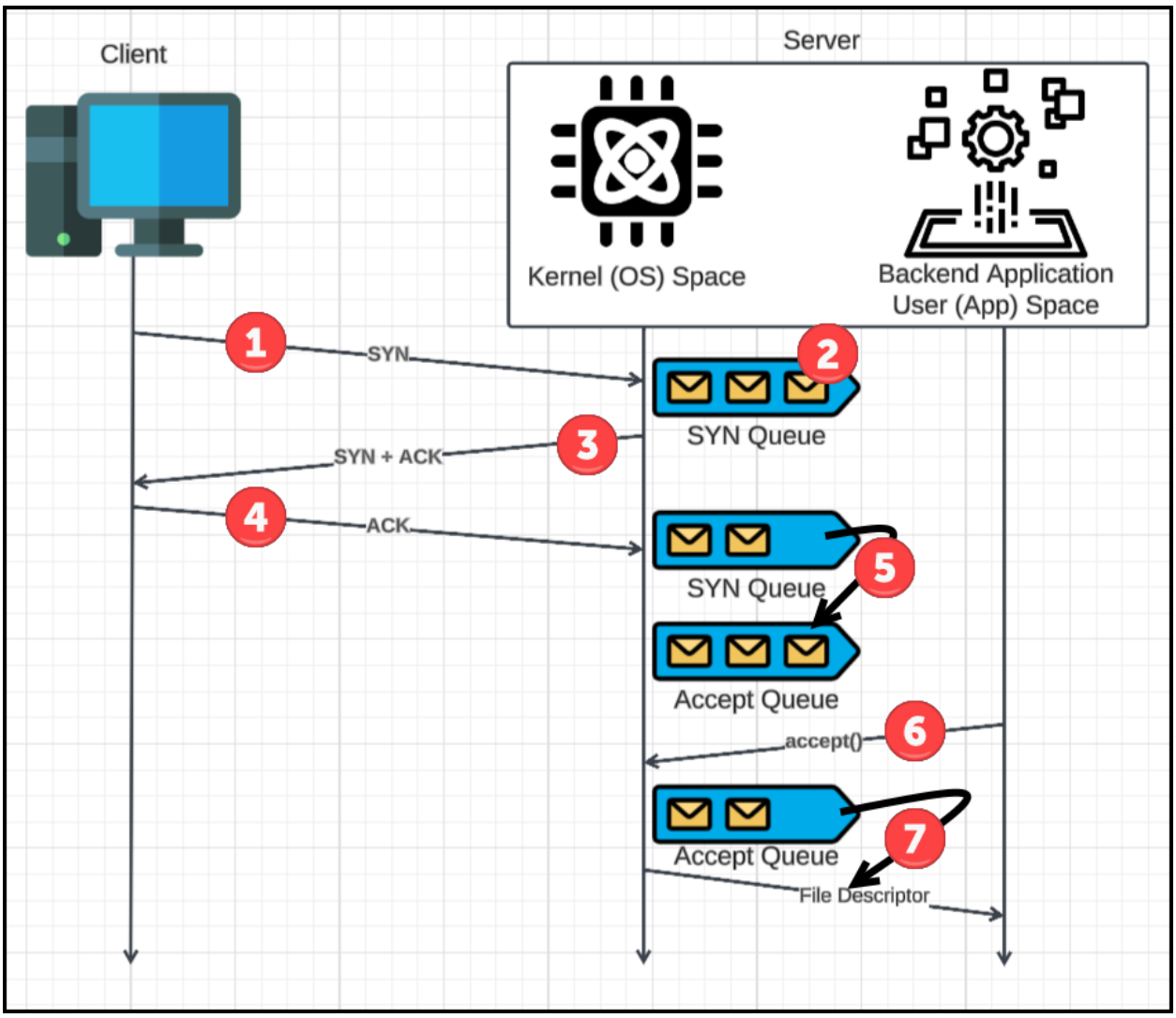
    - **User Space:** Your application process runs in "user space" with its own dedicated memory.
    - Understanding this separation is key to optimizing backend application performance.
- **The `listen()` Call:** When your application calls `listen()` on an address and port, it's telling the kernel: "Hey, I want you to handle incoming connection requests for me on this address/port."
- **Kernel Handles Handshake:** The **kernel**, not your application, performs the TCP three-way handshake (SYN, SYN-ACK, ACK).
- **Sockets and File Descriptors:**
    - When the kernel starts listening, it creates a "listening socket." This socket is a logical endpoint identified by a **file descriptor** (a small integer, like a pointer to the connection in the kernel).
    - **Socket vs. Connection:** A **socket** is the listening point (like a wall socket), while a **connection** is an instance created *from* that socket when a client successfully connects (like a device plugged into the wall socket). One listening socket can handle hundreds or thousands of individual connections.

The Kernel's Connection Queues

The kernel maintains **two important queues** for incoming TCP connections:

1. **SYN Queue (or Incomplete Connection Queue):**
    - **Purpose:** Holds information about incoming SYN requests that have been received, and the kernel has sent a SYN-ACK, but the client hasn't yet responded with the final ACK.
    - **Process:**
        - Client sends SYN.
        - Kernel adds the SYN (with source IP, source port, etc.) to the SYN queue.
        - Kernel replies with SYN-ACK.
        - The entry stays in this queue until the client's ACK is received.

- **Defence against SYN Flooding:** If an attacker sends many SYN requests but never ACKs, this queue can fill up, preventing legitimate connections. Modern kernels use SYN cookies to mitigate this by making the SYN-ACK stateless, avoiding filling the queue until the final ACK arrives.



2. **Accept Queue (or Completed Connection Queue):**
   - **Purpose:** Holds full, established TCP connections that have completed the three-way handshake and are ready for the application to "accept."
   - **Process:**
     - Client sends ACK.
     - Kernel receives ACK, matches it to an entry in the SYN queue.
     - Kernel removes the entry from the SYN queue.

        - Kernel moves the now fully established connection to the Accept queue.
        - A new file descriptor is created for this specific connection.
- **`backlog` Parameter:** Both the SYN queue and Accept queue have a finite size, often controlled by a `backlog` parameter when calling `listen()` (though high-level languages like Node.js might not directly expose this). A small `backlog` can cause new connections to be rejected if the queues fill up.

Application's Role: The `accept()` Call

Your backend application becomes involved once a connection is fully established in the Accept queue.

- **`accept()` System Call:** Your application calls `accept()` (or an equivalent function in your programming language/framework).
- **Moving Connection to User Space:**
    - `accept()` retrieves a ready connection from the Accept queue.
    - The kernel "hands over" this connection to your application.
    - The connection data is moved from kernel memory to your application's user space memory.
    - Your application receives the **file descriptor** for this specific connection.
- **Reading/Writing:** Now, your application can use this file descriptor to read data from the client or write data back to the client.
- **Blocking `accept()`:** If your application calls `accept()` but the Accept queue is empty, the `accept()` call will typically **block** (pause your application's execution) until a new connection becomes available. This is why asynchronous I/O (Input/Output) models are crucial for high-performance servers, allowing your application to do other work while waiting for connections.

2. Reading and Writing Data on TCP Connections

Once a TCP connection is established (as we discussed with the three-way handshake), the real work of exchanging data begins.

Understanding Send and Receive Buffers

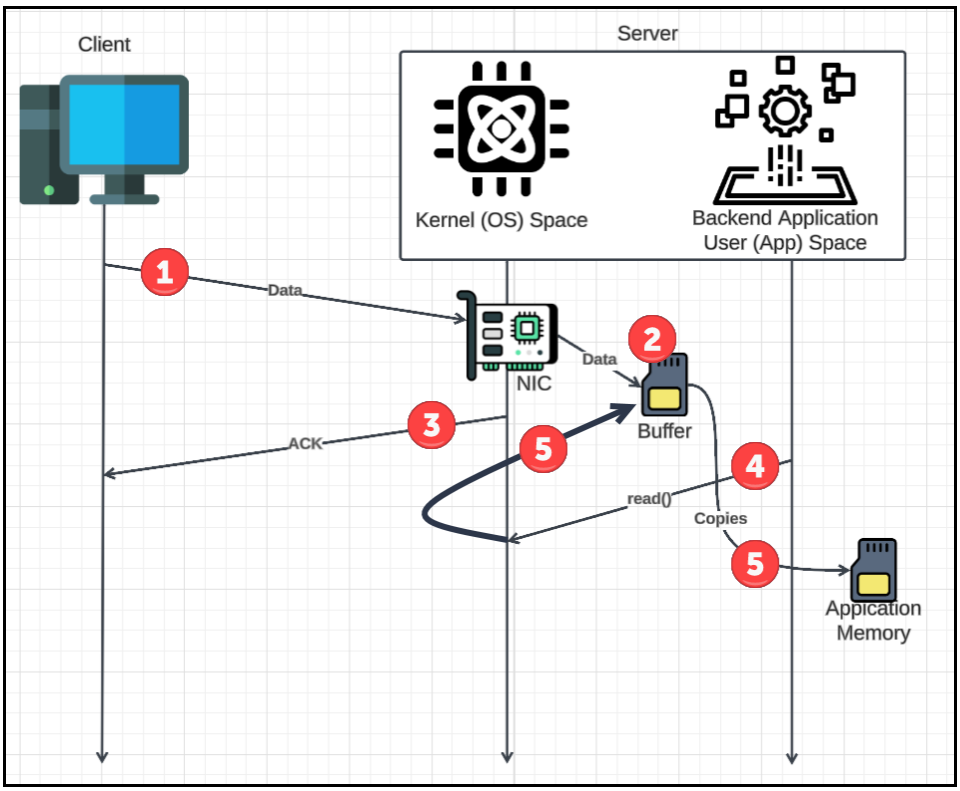
- Beyond the **SYN** and **Accept** queues, the **kernel** also manages two types of **buffers** for each established TCP connection:
    - **Receive Buffer:** A memory area in the kernel where incoming data from the network is stored *before* your application reads it.
        - Filled automatically by the kernel as packets arrive.
        - Your app calls `read()` to extract data from this buffer.
    - **Send Buffer:** A memory area in the kernel where data your application wants to send is stored *before* the kernel transmits it over the network.
        - Your app calls `send()` or `write()` to place data into it.
- These buffers are managed by the kernel on behalf of your application, greatly simplifying network programming.

How Data is Received by the Server (Client Sends to Server)

When a client sends data to your backend application:

1. **Client Writes Data:** The client application writes data to its side of the TCP connection (to its socket's file descriptor).
2. **NIC and OS Processing:**
    - The **Network Interface Card (NIC)** on the server receives the raw network frame.
    - The kernel then processes this frame:
        - It extracts the **IP packet**.
        - It then extracts the **TCP segment** (which contains the port number).
    - The kernel identifies which established connection (and thus which application process) this data belongs to.

3. **Data into Receive Buffer:** The kernel takes the incoming data and places it directly into the dedicated **Receive Buffer** for that specific connection. Your application hasn't even asked for it yet; the kernel proactively buffers it.



4. **Kernel Acknowledges Data:**
    - After placing data into the Receive Buffer, the kernel sends a **TCP ACK (Acknowledgment)** back to the client. This tells the client that the data has been successfully received by the server's kernel.
    - The kernel also updates the **flow control window size**, informing the client how much more data the server can receive before its buffer fills up.
    - **Delayed Acknowledgment (Delayed ACK):** To improve efficiency and reduce network overhead, the kernel might delay sending ACKs. Instead of acknowledging every single byte, it might wait to receive more data or for a short period of time before sending a single ACK for multiple received segments.
5. **Consequences of a Full Receive Buffer:**
    - The Receive Buffer has a finite size.

- If your backend application doesn't **read data fast enough** from this buffer, it will fill up.
- When the Receive Buffer is full, the kernel's reported window size will become zero, telling the client to stop sending data.
- If the client ignores this or sends data before the window updates, the kernel will **drop new incoming packets** because there's no space to store them. This leads to performance degradation and client slowdowns.

How the Application Reads Data (Server Reads from Receive Buffer)

Your backend application needs to actively pull data from the kernel's Receive Buffer:

1. **Application Calls read():** Your application (or its underlying framework/library) makes a system call like read() on the connection's file descriptor.
2. **Data Copying:** The kernel then **copies** the data from its Receive Buffer (in **kernel space**) into your application's memory (in **user space**).
    - **The Cost of Copying:** This copying operation, while fast (microseconds), can become a significant performance bottleneck in high-throughput applications, as it consumes CPU cycles and memory bandwidth.
    - **Zero-Copy (Advanced Optimization):** Newer kernel features (like io_uring in Linux) are being developed to enable "zero-copy" operations, where data isn't physically copied but instead pointers or shared memory are used, minimizing overhead.
3. **Blocking read():** If your application calls read() and the Receive Buffer is empty, the read() call will typically **block**. This means your application's execution will pause until new data arrives in the buffer.
    - **Asynchronous I/O Models:** For high-performance servers, this blocking behavior is undesirable. Modern backend frameworks use **asynchronous I/O models** (like epoll, select, kqueue) where the application tells the kernel: "Notify me when data is ready on any of these connections," instead of blocking. This allows the application to handle multiple connections concurrently without freezing.

4. **Application-Level Processing (After Kernel's Job):** Once the raw, encrypted bytes are copied into your application's memory, the kernel's primary job for receiving data is done. Now, your application's libraries take over:
    - **TLS Decryption:** If HTTPS is used, the **TLS library** (e.g., OpenSSL, LibreSSL) in your application decrypts these raw bytes using the symmetric key established during the TLS handshake. This is a CPU-intensive operation.
    - **Protocol Parsing:** The **HTTP library** (e.g., in Node.js Express, Python Flask) then parses the decrypted bytes into a meaningful HTTP request (e.g., identifying GET/POST, headers, body).
    - **Object Creation:** Finally, the library creates high-level, developer-friendly objects (like a request object) that contain parsed headers, body, and other request details, which your application code then consumes.

How Data is Transmitted by the Server (Server Sends to Client)

When your backend application sends a response back to the client:

1. **Application Writes Data:** Your application calls a function like response.write() or response.end() to send data.
2. **TLS Encryption & OS send():**
    - The **TLS library** in your application encrypts the response data using the symmetric key.
    - The application then makes a system call like send() to the operating system, passing the raw encrypted bytes.
3. **Data to Send Buffer:** The kernel **copies** the data from your application's memory (user space) into its **Send Buffer** (kernel space). Again, this copying incurs a cost.
4. **Kernel Buffering and Nagle's Algorithm:**
    - The kernel buffers the data in the Send Buffer. It doesn't necessarily send it immediately.
    - **Nagle's Algorithm:** By default, TCP employs Nagle's algorithm. This algorithm aims to improve network efficiency by:
        - Delaying the sending of small outgoing TCP segments.

        - Waiting until enough data accumulates to fill a full **Maximum Segment Size (MSS)** packet, or until the previous ACK for sent data is received.
        - **Why?** Sending many small packets is inefficient because each TCP/IP packet carries significant header overhead (around 40 bytes) regardless of data size. Nagle's algorithm tries to minimize this "header tax."
    - **Efficiency vs. Latency:** Nagle's algorithm prioritizes network efficiency over immediate data delivery. For interactive applications (like chat, gaming, or certain APIs), this small delay can be problematic.
    - **Disabling Nagle's:** Developers can often disable Nagle's algorithm (e.g., using the TCP_NODELAY socket option) if **low latency** is more critical than network efficiency for a specific use case.
5. **Asynchronous Sending:** The send() call is generally asynchronous. When your application calls send(), it means the data has been copied to the kernel's Send Buffer, *not* that it has been sent over the wire yet. The kernel will handle the actual transmission in the background.
6. **Client Acknowledgment:** Once the client receives and acknowledges the data, the kernel can safely remove that data from its Send Buffer.

3. Listener, Acceptor, and Reader

Building on our understanding of how the Linux kernel handles TCP connections, including the SYN and Accept queues, and the process of reading/writing data, this section introduces key architectural concepts in backend applications. We'll define the roles of **Listeners, Acceptors, and Readers** and see how these responsibilities can be managed within a backend system.

The Listener

- **What it does**: Calls the `listen()` system call on a given `IP + Port`.
- **Output**: A socket file descriptor (`FD`), representing a listening socket.
- **Responsibility**: Just creates the socket and tells the kernel to listen.
- **Typical Code**: `socket()`, `bind()`, `listen()`

The Acceptor

- **What it does**: Calls `accept()` on the listening socket `FD`.
- **Output**: A new file descriptor for each fully established connection.
- **Responsibility**: Pulls connections from the kernel's Accept Queue.

The Reader

- **What it does**: Uses `read()` to get data from the connection `FD`.
- **Responsibility**: Reads client requests and handles them (e.g., parses HTTP).
- **Can also**: Call `write()` or `send()` to send responses back.
- **Job Summary:** The Reader's job is to **process incoming data** from clients. It takes the raw bytes received from the kernel, decrypts them (if TLS/HTTPS is used), parses them into meaningful requests (like HTTP requests), and then passes them to the application's business logic. (Note: A corresponding **Writer** would handle sending data back to the client, using `send()` calls and filling the kernel's Send Buffer.)

Single Listener, Single Worker Thread

In this architectural pattern, a single process (which runs in a single thread) is responsible for all aspects of managing network connections:

- **Single Listener:** This one process handles the `listen()` call, binding to the specified IP address and port, and creating the listening socket.
- **Single Acceptor:** It's also this same single process that calls `accept()` to pull newly established connections from the kernel's **Accept Queue**.
- **Single Reader/Writer:** Furthermore, this single process is responsible for reading incoming data from all established connections (from the kernel's **Receive Buffers**) and writing outgoing data (to the kernel's **Send Buffers**).

Essentially, one thread does *everything* related to network I/O.

How It Works (Conceptual Flow)

1. Client sends a request.
2. Kernel (OS) receives the TCP packet.
3. Packet goes through:

    - SYN Queue → during TCP handshake
    - Accept Queue → once handshake completes
4. The process:
    - Has already called `listen()` → kernel prepared socket
    - Calls `accept()` internally to grab new connection
    - Starts reading from the connection using async events
5. Process uses epoll (Linux readiness-based mechanism) to watch many file descriptors:
    - Tells OS: "Let me know when one of these sockets is ready to read."
    - When data arrives → Process reads it via callbacks

Single Listener + Multiple Reader Threads

This section describes another common backend architectural pattern that aims to improve performance and scalability by leveraging multiple CPU cores. It's known as the **single listener, multiple worker threads** model. Applications like memcached often employ variations of this architecture.

In this architecture, a single main process (which might be single-threaded itself for the listening/accepting part) is responsible for managing incoming connections, but the actual data processing is offloaded to a pool of separate "worker" threads.

- **Single Listener and Acceptor (Often Combined in Main Process):**
    - A single main process or thread is dedicated to calling `listen()` and `accept()`.
    - This main process maintains an infinite loop, constantly pulling newly established connections from the kernel's **Accept Queue**.
    - The `accept()` operation is handled by this single entity.
- **Multiple Worker Threads:**
    - The main process (the acceptor) does **not** process the actual data itself.

    - Instead, it has already spawned (or "spun up") a fixed number of **worker threads**. This number often corresponds to the number of CPU cores available on the machine, allowing the application to utilize the hardware efficiently.
    - Each worker thread is dedicated to reading and processing data from client connections.
- **Connection Hand-off / Load Balancing:**
    - When the main acceptor process pulls a new connection from the Accept Queue, it then **hands off** this connection (its file descriptor) to one of the available worker threads.
    - This "hand-off" often involves some form of load balancing to distribute connections evenly among the worker threads.

How It Works

1. Server starts up and binds to a socket (IP + Port).
2. One **main thread** acts as both the **Listener** and the **Acceptor**.
3. The main thread accepts each incoming connection and:
    - Selects a worker thread from a pool
    - Passes the connection to that worker (e.g., using a queue or assignment logic)
4. Worker threads are already spun up and assigned to:
    - Read from the connection
    - Possibly process the request

Single Listener with Request-Level Load Balancing

1. The Challenge with Previous Models

- In the "Single Listener, Multiple Worker Threads" model, while connections were distributed, the actual workload on each connection could be uneven. A single "greedy client" or a connection with many active streams could pin a worker thread, leaving others idle and leading to overall system bottlenecks.

2. RamCloud's Approach: Request-Level Load Balancing

- RamCloud, a high-speed storage system for data centers, employs an architecture that addresses this problem by moving the load balancing point to a higher level: at the request/message level.
- Core Idea: Instead of handing off raw connections to worker threads, the main process (listener/acceptor) handles all the initial network I/O, including decrypting and parsing the request. Only then is the fully formed, logical request (e.g., an HTTP request or a gRPC message) dispatched to a worker thread.

The Architecture Breakdown:

- **Single Listener & Acceptor (Main Process):**
  - There is still a single main process that calls `listen()` and `accept()`.
  - This process continually pulls new connections from the kernel's **Accept Queue**.
  - This main process is also responsible for reading raw data from these connections from the kernel's **Receive Buffers**.
- **"Protocol-Aware" Processing (Within the Main Process):**
  - This is the critical difference: The main process doesn't just read raw bytes. It performs the necessary preliminary work:
    - **TLS Decryption:** If HTTPS/TLS is used, the main process decrypts the incoming raw bytes.
    - **Protocol Parsing:** It then parses these decrypted bytes into a complete, logical "request" or "message" based on the application protocol (e.g., HTTP, gRPC).
  - This main process needs to be "protocol aware" to correctly interpret the incoming stream of bytes. This implies a very low-level understanding and implementation of the network stack.

- **Dedicated Worker Threads for Request Execution:**
    - Once a complete, parsed, and decrypted request/message is formed, it is then placed into a queue or directly dispatched to one of a pool of dedicated **worker threads**.
    - These worker threads are now "clean worker threads" because their sole responsibility is to **execute the actual application logic** associated with that specific request. They don't deal with raw network I/O, decryption, or parsing.

Advantages of This Model

- **True Load Balancing:** Because worker threads receive *parsed requests* rather than raw connections, the work can be truly load-balanced. If one request is computationally heavy and takes longer, only one worker thread is tied up, while others can continue processing other ready requests. This prevents the "greedy client" problem observed in the previous model.
- **Optimal CPU Utilization:** This approach helps ensure that all worker threads are maximally utilized, leading to more consistent performance and higher overall throughput.
- **Worker Thread Simplicity:** Worker threads are simpler, as they don't need to handle complex network I/O, buffering, or protocol parsing. Their logic is focused solely on fulfilling the request.

How It Works

1. **Listener** creates the socket and begins listening.
2. **Acceptor** thread picks up new connections.
3. Connection threads:
    - Read incoming raw data (TCP packets)
    - Decrypt (if encrypted)
    - Parse protocol (e.g., HTTP, gRPC)
    - Assemble a clean request message
4. The **request message** is passed to a **Worker** Thread.
5. Worker thread executes the actual job (e.g., lookup, write, response).

Multiple Threads with a Single Socket

This section explores another architectural approach for backend systems: **multiple threads operating on a single listening socket**. This model, notably used by **Nginx** (though its default configuration has changed), aims to improve concurrency in accepting new connections.

The Architecture: Multiple Threads, One Socket

- **Core Idea:** In this pattern, there's effectively a single "listening socket" created by a main process or thread, but multiple worker threads within the *same process* are configured to all try to `accept()` connections from this *one shared listening socket*.
- **Shared Memory:** Because all these threads belong to the same process, they share the same memory space. This means they all have direct access to the single listening socket object/file descriptor.

How it Works (with Nginx as a Historical Example)

1. **Single Listener (Main Process/Thread):** A primary part of the application (e.g., Nginx's master process) calls `listen()` on a specific address and port, creating the single listening socket.
2. **Multiple Worker Threads:** The application then spawns multiple worker threads. Each of these worker threads is configured to directly attempt to `accept()` connections from that *same shared listening socket*.
3. **Concurrent `accept()` Calls:** Each worker thread continuously tries to pull a new connection from the kernel's **Accept Queue** by calling `accept()` on the shared socket.

The Challenge: Mutex and Contention

- **Race Condition:** You cannot have multiple threads simultaneously `accepting()` from the *exact same* underlying socket object without coordination. This would lead to a "race condition" where threads "stomp on each other's toes" trying to grab the next available connection.
- **Accept Mutex (Nginx Example):** To prevent this, such architectures typically employ a **mutex** (short for "mutual exclusion").

    - A mutex is a synchronization mechanism that ensures only one thread can access a shared resource (in this case, the `accept()` call on the shared socket) at a time.
    - For example, Nginx historically used an `accept_mutex`. Before a thread can call `accept()`, it must acquire this mutex. If another thread already holds the mutex, the current thread must wait. Once it's done accepting, it releases the mutex, allowing another thread to acquire it.
- **Implication:** While having multiple threads trying to `accept()` is conceptually good, the mutex introduces **locking and unlocking overhead**. This means that even with multiple threads, only one connection can truly be accepted at a given instant from that single socket.

Multiple Listeners on the Same Port (Socket Sharding)

This section introduces a highly optimized backend architectural pattern: **multiple processes or threads listening on the same port**. This technique, often called **socket sharding**, is widely used by high-performance proxies and web servers like Nginx and Envoy.

The Default Behaviour: Port Conflict

Normally, if one process is listening on a specific IP address and port (e.g., 192.168.1.5:8080), another process trying to listen on the *exact same* IP address and port will fail with an "Address already in use" error. This prevents two applications from trying to claim the same network resource.

The Exception: `SO_REUSEPORT` (Socket Reuse Port)

- **Enabling Sharing:** Modern operating systems (especially Linux) provide a socket option called `SO_REUSEPORT` (or similar options like `SO_REUSEADDR` which has slightly different implications) that allows **multiple independent processes or threads to listen on the exact same IP address and port simultaneously.**
- **The "Neat Trick": Kernel-Level Load Balancing:**
    - When `SO_REUSEPORT` is enabled, the operating system's kernel becomes a **load balancer** for incoming connections.

- Instead of just one pair of `SYN` and `Accept` queues, the kernel effectively creates **separate pairs of `SYN` and `Accept` queues for each process** that is listening on that shared port.
- When a new `SYN` request arrives from a client, the kernel uses a **hashing algorithm** (often based on the client's source IP and port, and destination IP and port) to intelligently distribute the incoming connection to one of the available listening processes' queues. This distribution is typically very efficient, similar to a round-robin approach.

Distributed Listening, Accepting, and Reading

With `SO_REUSEPORT`, the architecture looks like this:

- **Multiple Independent Processes/Threads:** You can spawn `N` number of backend processes (or threads, if the OS supports it for threads specifically) where each one:
  - Acts as its own **Listener** (calling `listen()` with `SO_REUSEPORT`).
  - Acts as its own **Acceptor** (calling `accept()` on its *own unique socket file descriptor*).
  - Acts as its own **Reader/Writer** (handling data for the connections it accepted).
- **No Mutex Contention:** The key advantage here is that there is **no need for an `accept_mutex`**. Each process/thread operates on its *own* unique listening socket (even though they all bind to the same `IP:Port`), and the kernel handles the initial distribution. This eliminates the contention bottleneck seen in the "multiple threads on a single socket" model.
- **True Parallelism:** Each process/thread can independently accept, read, and write data without blocking others, leading to true parallelism and better utilization of multi-core CPUs.

4. Advantages of Socket Sharding

- **Eliminates Accept Bottleneck:** Solves the contention issue of a single `accept()` `lock`, allowing connection acceptance to scale across multiple cores.

- **Kernel-Level Load Balancing:** The OS kernel, which is highly optimized for performance, handles the initial distribution of connections, ensuring efficient load balancing.
- **Scalability:** Allows backend applications to easily scale horizontally by simply spinning up more processes/threads that listen on the same port.
- **Robustness:** If one process crashes, the others can continue accepting connections.
- **Popular Choice:** This is almost the default architecture for high-performance proxies and load balancers like Nginx and Envoy due to its efficiency.