Application Layer Architecture and HTTP

Contents

[Agenda 2](#_Toc185257475)

[Application Layer Architectures 2](#_Toc185257476)

[Peer-to-Peer (P2P) Architecture 5](#_Toc185257477)

[Application Layer Architectures - Client-Server and Peer-to-Peer 8](#_Toc185257478)

[How Applications Connect to a Network 11](#_Toc185257479)

[Ports, Sockets, and Communication in Networking 14](#_Toc185257480)

[Ports, Sockets, and Communication in Networking 16](#_Toc185257481)

[HTTP Protocol 20](#_Toc185257482)

[HTTP and Stateless Protocol 23](#_Toc185257483)

[HTTP Protocol Overview 26](#_Toc185257484)

[HTTP Methods and Search API Design 29](#_Toc185257485)

[Key Topics Discussed 31](#_Toc185257486)

## Agenda

1. **Application Layer Architectures**
   * Overview of two types:
     + Client-Server Architecture.
     + Peer-to-Peer (P2P) Architecture.
   * Brief discussion on BitTorrent and its working.
2. **Sockets vs. Ports**
   * What are Sockets?
   * What are Ports?
3. **HTTP Protocol**
   * HTTP Requests and Responses.
   * Properties of HTTP (e.g., Cookies).

## Application Layer Architectures

* **Definition**:
  + Architectures define how applications at the application layer are structured and interact.
* **Two Main Architectures**:
  + **Client-Server Architecture**
  + **Peer-to-Peer (P2P) Architecture**
  1. **Introduction to the Application Layer**
* **Purpose**:
  + **At both ends of data**: Responsible for **creating** and **receiving data**.
  + Serves as the **endpoints** of communication, interacting directly with users.
  + Enables applications to **write data to the network** and **read data from the network**.
* **Significance**:
  + Internet's true impact comes from the **plethora of applications** it supports (e.g., social media, chatting apps, World Wide Web).
  + Without these applications, the Internet itself would not have revolutionized the world.
* **Analogy**:
  + In a **postal network**, the application layer is equivalent to the **human** who sends or receives a letter.
  1. **Client-Server Architecture**
* **Concept**:
  + Communication happens between two distinct entities:
    1. **Client Application**:
       - A front-end application requesting data.
       - Examples: Browser UI (React apps, HTML/CSS/JavaScript).
    2. **Server Application**:
       - A back-end application providing data.
       - Examples: Applications built using Spring Boot, Django, Golang, or Node.js.

A diagram of a computer

Description automatically generated

* **How it Works**:
  + Client sends a **request** for data.
  + Server processes the request and sends back a **response**.
* **Key Features**:
  + **Two Different Applications**:
    1. Front-end and back-end are separate entities communicating via messages.
  + Examples in Programming:
    1. **Client**: React-based front-end code.
    2. **Server**: Spring Boot, Django, or Node.js-based back-end code.
  1. **Importance of Client-Server Separation**
* Both client and server are distinct but related.
* They interact via **message passing**.

## Peer-to-Peer (P2P) Architecture

**Definition**

* A decentralized communication model where the same application runs on all devices involved in communication.
* Each node acts as both **sender** and **receiver**, enabling data exchange between peers.

**Key Characteristics**

1. **Decentralized**: No central server; all nodes share responsibility for communication and data exchange.
2. **Same Codebase**: The same application (e.g., uTorrent) runs on every participating device.
3. **Dynamic Roles**: A node can send or receive data depending on its needs.

A diagram of a computer network

Description automatically generated

**Example: Torrent Systems**

* Torrent systems epitomize P2P architecture.
* File distribution involves breaking files into small parts shared across nodes.

**How Torrent Systems Work**

1. **Torrent File**
   * A small file (1–2 KB) containing metadata about the actual content.
   * Includes URLs of **tracker servers**.
2. **Tracker Servers**
   * Maintain a list of IP addresses and the file parts those nodes possess.
   * Facilitate initial peer discovery and data exchange coordination.
3. **File Sharing Process**
   * Files are divided into multiple parts.
   * Nodes request missing parts from peers with the desired data.
   * As a node acquires parts, it becomes a source for other peers.

**Behaviour During Downloading**

* Initially, nodes download file parts without sharing.
* Once some parts are acquired, nodes start uploading these parts to other peers.
* Torrent systems track **upload** and **download speeds** to ensure fair sharing.

**Key Terms in Torrent Systems**

1. **Seeder**
   * A node that has downloaded the complete file and continues to upload it for others.
   * Encourages file availability and network growth.
2. **Leecher**
   * A node downloading the file but not uploading any parts to others.
   * Typically reduces overall network efficiency.
   * Torrent systems often throttle the download speed of leechers to discourage this behaviour.

**Additional Points**

1. **Dynamic Updates**
   * Trackers dynamically update the availability of parts as nodes join or leave the network.
2. **Upload-Download Balance**
   * Torrent systems maintain a balance between uploading and downloading to ensure smooth functioning and limit freeloading.

**Analogies and Examples**

* **Seeders**: Like seeds growing plants, they spread the file across the network.
* **Leechers**: Like parasites, they consume resources without giving back.

**Advantages of P2P Architecture**

1. Decentralized data sharing increase’s reliability and fault tolerance.
2. Scalability due to distributed load.

**Challenges**

1. Dependence on participants for availability.
2. Potential for misuse, as seen in unauthorized file sharing.

## Application Layer Architectures - Client-Server and Peer-to-Peer

**1. Client-Server Architecture**

* **Definition**:
  + A structured model where two distinct entities exist: a **client** and a **server**.
  + Responsibilities are **divided**:
    - **Client**: Requests data or services.
    - **Server**: Provides data or services.
* **Characteristics**:
  + **Centralized architecture**: The server acts as the central point for storing and sharing data.
  + **Examples**:
    - Websites (e.g., streaming platforms like Spotify post-2014).
    - Traditional database systems.
* **Pros**:
  + Centralized control ensures security and easy data management.
  + Works well for predictable, large-scale, organized communication.
* **Cons**:
  + Single point of failure (if the server goes down, the system stops functioning).
  + Scalability can be an issue under heavy loads.

**2. Peer-to-Peer (P2P) Architecture**

* **Definition**:
  + A distributed model where all devices (peers) run the **same application** and share equal roles in communication.
  + Peers can act as both **sender (Seeder)** and **receiver (Leecher)** simultaneously.
* **Characteristics**:
  + **Decentralized**: No central server; all peers participate in data sharing.
  + Data is often split into smaller **chunks** and distributed across multiple peers.
  + **Dynamic topology**: Peers can join or leave the network at any time.
* **Examples**:
  + **Torrents**:
    - File-sharing systems like uTorrent, BitTorrent.
    - Torrent files contain **tracker server** information, which maps parts of the file to IP addresses.
    - Seeders: Fully downloaded the file and are uploading parts to others.
    - Leechers: Downloading but may not upload or contribute.
  + Legacy Applications:
    - Skype (pre-2012).
    - Spotify (pre-2014).
    - Cryptocurrency systems like Bitcoin.
* **Pros**:
  + **Scalability**: More peers improve network performance.
  + Resilient to individual peer failures.
* **Cons**:
  + Security concerns: Decentralized structure can be exploited.
  + Reliability depends on peer availability and behavior.

**3. Comparison: Client-Server vs Peer-to-Peer**

| **Feature** | **Client-Server** | **Peer-to-Peer** |
| --- | --- | --- |
| **Responsibility** | Divided between client & server | Shared among all peers |
| **Control** | Centralized | Decentralized |
| **Scalability** | Limited under heavy loads | High, as peers increase |
| **Failure Impact** | Server failure stops system | Peer failure has minimal impact |
| **Examples** | Websites, cloud services | Torrents, blockchain |

**4. Torrent System Functionality**

* **Step-by-step process**:
  1. Download a **torrent file** (small file containing tracker server information).
  2. The application (e.g., uTorrent) retrieves IP addresses and data locations from the tracker servers.
  3. Files are divided into chunks, and each peer downloads chunks from other peers.
  4. Once a peer has chunks, it uploads them to others while continuing its downloads.
* **Key Terms**:
  1. **Seeders**: Fully downloaded the file but continue uploading to share with others.
  2. **Leechers**: Downloading files and may or may not contribute by uploading.
* **Behaviour**:
  1. Torrent systems prioritize fairness by reducing download speed for leechers who do not upload.
  2. Increased seeders result in faster downloads.

**5. Modern Adaptations**

* Applications like Spotify and Skype transitioned from **P2P** to **client-server** models for:
  + Better control over content distribution.
  + Enhanced user experience with centralized servers.
  + Improved security and data handling.

**Key Takeaways:**

* **Client-Server** and **Peer-to-Peer** architectures cater to different needs and scales.
* P2P systems shine in **distributed and resilient** scenarios like file-sharing or blockchain, whereas client-server architectures excel in **centralized control** scenarios like web services.
* Real-world applications evolve, adapting architectures to meet performance, security, and scalability demands.

## How Applications Connect to a Network

**Overview**

* Applications connect to networks through **sockets**, which serve as interfaces between the application and the operating system.
* **Sockets** are objects provided by the operating system API (e.g., Linux, Windows, macOS) that allow data exchange over a network.

**Key Concepts**

1. **Role of Operating System**:
   * Operating systems provide APIs to create and manage sockets.
   * These APIs are low-level and form the foundation of network communication.
2. **Socket as an Object**:
   * Sockets are analogous to objects used for interacting with files (e.g., File in Java).
   * Every network interaction—sending, receiving, or connecting—requires a socket object.
   * Applications create socket objects via operating system APIs (e.g., socket()).
3. **High-Level Abstractions**:
   * Libraries like Python's requests, Java's HTTP clients, or tools like curl abstract socket creation.
   * These libraries internally utilize the operating system's socket API.
4. **Sockets at Both Ends**:
   * Both the client and the server create socket objects for communication.
   * Each socket is associated with a unique **port number** for identification.

**Detailed Workflow**

1. **Socket and Port Mapping**:
   * When an application creates a socket, it is assigned a **port number** by the operating system.
   * The operating system maps ports to sockets to route data to the correct application.
2. **Communication Steps**:
   * The client uses the server’s IP address and port to send a request.
   * The operating system at the server end identifies the socket associated with the port and forwards the data to the application.
   * The server responds back to the client’s IP address and port, following a similar mechanism.
3. **Ports on the Client Side**:
   * Clients also have ports, which are dynamically assigned during connection establishment.
   * These ports are essential for receiving responses from the server.
4. **Default Protocol Ports**:
   * Common protocols have default ports (e.g., HTTPS: 443, HTTP: 80).
   * Specifying the port is unnecessary unless the server uses a non-standard port.

**Practical Examples**

* **Client-Server Interaction**:
  + A client opens a socket to send a request to a server.
  + Example:
    - Client: IP1, Port 12345
    - Server: IP2, Port 443 (default HTTPS port)
* **Libraries Abstract Socket Operations**:
  + Python: requests.get(url) internally creates and uses sockets.
  + Command-line tools like curl also use sockets under the hood.

**Special Cases and Questions**

1. **Can applications create multiple sockets?**
   * Yes, each socket must be associated with a unique port.
2. **Why are sockets required at both ends?**
   * A socket is the interface that allows the operating system to deliver incoming data to the correct application.
3. **How does the OS handle incoming data?**
   * The operating system maps incoming data (based on the port) to the corresponding socket and delivers it to the application.
4. **Can one backend application use multiple ports?**
   * Yes, this is achieved by creating multiple socket objects.

**Key Takeaways**

* **Socket**:
  + Core object for network communication.
  + Acts as the bridge between an application and the network.
* **Port**:
  + Identifier used by the operating system to map incoming and outgoing data to the correct socket.
* **Operating System API**:
  + Provides the mechanisms for socket creation and data routing.

## Ports, Sockets, and Communication in Networking

**1. Ports and Their Role**

* **Server Port**:
  + A server should always run on a **fixed port number** (e.g., Port 443 for HTTPS).
  + Fixed ports allow clients to reach the server consistently.
  + Servers do not assign random ports because users rely on known ports.
* **Client Port**:
  + The client uses **random ports** (ephemeral ports) when initiating a connection.
  + These ports are automatically assigned by the **Operating System** to an available port.
  + **Ephemeral ports**: Temporary ports assigned dynamically (range: **2^16** = 65,535 ports).

**2. Understanding Sockets**

* A **socket** is a combination of:
  + **IP Address + Port** (e.g., 192.168.0.1:443)
  + It enables communication between a client and server.
* **Socket Behaviour**:
  + On the server, one socket (e.g., Port 443) handles all incoming connections.
  + The server socket identifies incoming requests based on the **IP address** and **port number** of the client.
  + Example: Think of a socket as a **door** to a house. No need for 100 doors for 100 visitors; one door handles all.

**3. Parallel Requests and Connection Handling**

* **Client Side**:
  + Each client request creates a **new socket** with its **own ephemeral port**.
  + Example: If 5 parallel requests are made, 5 separate sockets are created.
* **Server Side**:
  + A single server port (e.g., 443) can serve multiple client requests simultaneously.
  + The server distinguishes each request using:
    - **Client IP address**
    - **Client ephemeral port**

A diagram of a computer network

Description automatically generated

**4. Port Limitations**

* Total number of ports = **2^16 = 65,535**.
* Each socket gets a unique port:
  + **Server port**: Fixed (e.g., 443)
  + **Client ports**: Ephemeral, dynamic allocation from the range of available ports.

**5. How Connections Work**

* Client sends data to the server using the **server's IP address and fixed port**.
* The server OS maps the incoming request to the **server socket**.
* The server processes the request and uses the **client IP and ephemeral port** to respond.

**6. Real-World Example: Google Chrome**

* Google Chrome handles multiple parallel connections:
  + **Each request** creates a **new socket**.
  + Example: A single tab can create multiple sockets (e.g., for analytics, media, etc.).

**Key Takeaways:**

1. Server ports are **fixed and remembered**; client ports are **random and ephemeral**.
2. One server socket can handle **many requests simultaneously**.
3. Sockets = Door to communication, allowing organized data exchange.
4. Port numbers are limited to 65,535; each connection (client-server) occupies one.

## Ports, Sockets, and Communication in Networking

**1. Difference Between Ports and Sockets**

* **Port**:
  + Acts as a **public identifier** (public address) for communication.
  + Example: **Port 443** is the default port for HTTPS.
  + It is **fixed** for servers to ensure consistency (e.g., Google uses port 443).
* **Socket**:
  + A **software construct** used to interact with the network.
  + Socket = **IP Address + Port**.
  + Example: 192.168.0.1:443 is a socket.
  + Think of a port as the **path to a file**, and a socket as the **object of the file**.

**2. Server and Client Communication**

* **Server**:
  + Runs on a **fixed port** (e.g., 443) to allow clients to connect consistently.
  + The server distinguishes connections using the **client IP address** and **client ephemeral port**.
* **Client**:
  + Uses a **random ephemeral port** when initiating a connection.
  + The OS dynamically assigns these ports to avoid conflicts.
* **Multiple Requests**:
  + A single server port (e.g., 443) can handle multiple client requests **simultaneously**.
  + Requests are identified by:
    - Client IP Address
    - Client ephemeral port
  + All requests use the **same server socket** but are managed internally.

**3. Parallel Requests and Thread Management**

* **Incoming Requests**:
  + Requests arrive at the **same socket** and are managed using **queues**.
  + Requests are processed **one at a time** or concurrently based on system capacity.
* **Application Threads**:
  + Servers use **thread pools** to manage concurrent requests.
  + Example:
    - If a server has **20 threads**, it can process **20 requests** simultaneously.
    - Remaining requests are queued until a thread becomes available.
* **Analogy**:
  + A **single door** (socket) allows multiple people (requests) to enter.
  + The **CPU** (dining table) processes requests one by one or in parallel based on its **thread pool capacity**.

**4. Load Balancing and Scalability**

* **Handling Large Traffic**:
  + Servers distribute incoming requests using **load balancers**.
  + Example:
    - Google runs multiple servers on **port 443** but with **different IP addresses**.
    - Load balancers ensure no single server becomes overwhelmed.
* **Port Capacity**:
  + Total ports:
  + If all ports are exhausted, the server **cannot accept new connections**.
  + **Solution**:
    - Use multiple servers to distribute requests and avoid port exhaustion.
    - For example, Scalar runs **90+ servers** to manage 10,000+ users.

**5. How Connections Work**

1. Client sends a request to a server using:
   * Server’s **IP Address** + **Port**.
2. Server maps the incoming connection using:
   * Client’s **IP Address** + **Ephemeral Port**.
3. Server responds to the client using the same mapping.

**6. Key Clarifications**

* **Can a Server Have Multiple Ports?**
  + Yes, a server can listen on **multiple ports** to handle different applications or processes.
* **Who Manages the Socket?**
  + **Operating System** creates and manages the socket.
  + The **process (application server)** uses the socket to handle incoming data.
* **What Happens if Ports are Exhausted?**
  + If all ports are used, the server **cannot create new sockets**.
  + Servers prevent this by:
    - Using load balancers.
    - Distributing requests across multiple servers.

**7. Real-World Examples**

* **Google**:
  + Multiple servers handle billions of requests.
  + Each server runs on **port 443** (HTTPS) but has a **different IP address**.
* **Thread Management**:
  + Server manages requests using **queues** and **thread pools**.
  + Example:
    - Quad-core machines can run multiple threads in parallel while queuing others.

**Summary**

1. **Port** = Public identifier (address) for communication.
2. **Socket** = Software construct; IP + Port combination.
3. **One Server Socket** can handle **multiple requests simultaneously**.
4. Servers use **thread pools** and **queues** to manage requests efficiently.
5. Load balancers ensure scalability by distributing requests to multiple servers.

## HTTP Protocol

**1. Introduction to HTTP**

* **HTTP** stands for **HyperText Transfer Protocol**.
* It is a **protocol** (set of rules) used to **transfer hypertext** (text enriched with formatting, images, links, etc.) across the internet.

**What is Hypertext?**

* Originally, the internet was used to send plain documents (just text).
* Hypertext enhanced plain text by including:
  + Bold, italic formatting
  + Images
  + Links, etc.
* **HTML (HyperText Markup Language):** Language used to create hypertext.
* **HTTP:** Protocol that allows the transfer of such hypertext.

**2. Modern Uses of HTTP**

* HTTP is no longer limited to hypertext:
  + Used for transferring files
  + Supports video calls, streaming, etc.
* HTTP is a general-purpose protocol for data transfer over the web.

**3. HTTP is a Client-Server Protocol**

* **Client:** Requests data from a server.
* **Server:** Responds with requested data.
* The communication process:
  + **Request:** Client sends a request for data.
  + **Response:** Server processes and sends back the data.

**4. HTTP and Transport Layer Protocols**

* **HTTP 1.x and HTTP 2:** Use **TCP** (Transmission Control Protocol) as the transport layer.
  + TCP ensures reliable delivery (guaranteed packet delivery, etc.).
* **HTTP 3:** Uses **UDP** (User Datagram Protocol) for transport.
  + **Why UDP?**
    - Faster and lightweight.
    - Internet reliability has improved, reducing packet loss concerns.
    - Efficient for applications requiring low latency.

**5. Stateless Nature of HTTP**

* **Stateless Protocol:** Each HTTP request is **independent** and **self-sufficient**.
  + Previous requests do not impact the next requests.

**Comparison with Stateful Protocol (FTP)**

* **FTP (File Transfer Protocol):**
  + Example: Login → Send File → Get File
  + **Stateful** because:
    - Later requests (e.g., Send File) depend on previous requests (e.g., Login).
* **HTTP:**
  + Each request must carry all the necessary information.
  + Example:
    - You log in to a website (scaler.com).
    - After login, subsequent requests (e.g., opening a dashboard) do not require re-sending the username/password.

**6. HTTP Cookies and State Management**

* HTTP is **stateless**, but applications need to manage user sessions.
* **Cookies:** Mechanism to maintain session state in a stateless protocol.
  + **How it works:**
    1. Client sends username/password in the first request.
    2. Server:
       - Authenticates the user.
       - Generates a **token** (unique identifier) for the user.
       - Stores the token on the server (e.g., in a database).
       - Sends the token back to the client.
    3. For future requests:
       - The client includes the token in every HTTP request.
       - The server validates the token to identify and authenticate the user.

**Key Points:**

* **Token Dependency:** Requests remain **independent** because:
  + The **token** itself carries the identification.
  + Any machine with the correct token can access the data.
* **Risk of Token Leak:** If the token is leaked, unauthorized users can access sensitive data.

**7. Summary of Key Concepts**

1. **HTTP**: Protocol for transferring hypertext and other data.
2. **Client-Server Model**: Client sends requests; server sends responses.
3. **Transport Layer**:
   * HTTP 1.x/2 → TCP
   * HTTP 3 → UDP
4. **Stateless Protocol**:
   * Each HTTP request is independent.
   * State is not preserved between requests.
5. **Cookies**:
   * Used to manage state in HTTP.
   * Tokens are generated to identify user sessions.

**8. Comparison Table**

| **Aspect** | **HTTP** | **FTP** |
| --- | --- | --- |
| **State Management** | Stateless | Stateful |
| **Dependency** | Independent requests | Dependent on previous actions |
| **Purpose** | Data transfer (web) | File transfer |
| **Transport Protocol** | HTTP 1.x/2 → TCP; HTTP 3 → UDP | TCP |

## HTTP and Stateless Protocol

**What is HTTP?**

* **HTTP** (HyperText Transfer Protocol) is a **stateless protocol**.
  + Stateless means **no session** is maintained between client and server.
  + Each request is independent of the previous one.

**Key Characteristics of HTTP**

1. **Statelessness**
   * **No memory** of previous requests.
   * Every request sent by the client must explicitly contain all the necessary details for the server to process it.
   * Example: Authentication tokens must be included in requests for the server to recognize the user.
2. **Scalability**
   * Stateless systems are easier to scale:
     + Requests can be routed to any server.
     + Servers can handle requests independently.
     + A common database stores session details, accessible to all servers.
   * Example: If Server A is overloaded, requests can go to Server B seamlessly.
3. **Tokens for Authentication**
   * After login, the server issues a **session token**.
   * The token is sent back to the client and stored (usually in cookies).
   * For subsequent requests, the client sends the token to authenticate.

**Session and Tokens**

* **Tokens** serve as an identifier for authenticated sessions.
* Tokens are usually stored in a **database** on the server side.
* Tokens can include:
  + Expiry dates (optional, implementation-specific).
  + User sessions (e.g., logged-in devices).

**How Stateless Authentication Works**

1. **Login Process**
   * User logs in → Server generates a token → Token sent to the client (browser).
2. **Token Storage**
   * Tokens are stored as **cookies** in the browser.
3. **Subsequent Requests**
   * Every new HTTP request includes the token in headers.
   * Server validates the token by checking the database.
4. **Session Removal**
   * Tokens can be invalidated or deleted (e.g., logging out from other devices).

**Advantages of Stateless Protocol**

1. **Easy Scalability**
   * Requests can be distributed to any server (load balancing).
   * All servers can access the same session database.
2. **Fault Tolerance**
   * If one server fails, requests can be redirected to another without data loss.
3. **Simplified Implementation**
   * Server does not need to maintain any in-memory session data.

**Example: Multiple Devices and Session Management**

1. **Login Example**
   * User logs in on multiple devices.
   * Server stores tokens in a **user session table** in the database.
2. **Session Limit**
   * Example: If only 3 sessions are allowed:
     + On a new login, the server checks for active sessions.
     + If the limit is exceeded, older sessions are logged out (tokens removed from the database).
3. **Session Validation**
   * Any request with an invalid token (e.g., deleted or expired) will result in the user being logged out.

**Why Not Use User ID/Password in Every Request?**

* Security risk: Sending credentials repeatedly increases chances of interception.
* Tokens are safer and more efficient for repeated authentication.

**Load Balancer**

* **Load balancers** help distribute requests to servers.
* Authentication can occur at the load balancer level before passing requests to servers.
* This improves efficiency and prevents unauthorized requests from reaching the servers.

**Comparison: Stateful vs Stateless Systems**

| **Aspect** | **Stateful Protocol** | **Stateless Protocol** |
| --- | --- | --- |
| **State Management** | Maintained on the server | No state maintained |
| **Scalability** | Difficult to scale | Easy to scale |
| **Example** | FTP | HTTP |
| **Session Persistence** | Session exists per server | Sessions are token-based |

**Key Takeaways**

1. HTTP does not maintain state – the client must provide all necessary information.
2. Tokens allow the server to identify users across stateless HTTP requests.
3. Stateless systems are more scalable and fault-tolerant.
4. Load balancers help distribute requests and validate sessions efficiently.

## HTTP Protocol Overview

* **Key Concept**: In the **web** or **Internet**, *everything is an object* or a file.
* Every object/file is stored at a **specific location** identified by its **URL**.

**Structure of a URL (Uniform Resource Locator):**

1. **Protocol**: HTTP, HTTPS, FTP, etc.
2. **Hostname/IP Address**: Example - scaler.com or 192.168.1.1
3. **Port**:
   * Default ports:
     + HTTP → 80
     + HTTPS → 443
4. **Path**: Location of the file or resource on the server.  
   Example: /meetings/I/XYZ
5. **Query Parameters**: Data sent in the URL after ?.  
   Example: utm\_source=abc&utm\_medium=xyz

**HTTP Request-Response Cycle**

* **Client-Server Model**:
  + **Client** sends an HTTP request to the server.
  + **Server** processes the request (via backend logic) and sends a **response**.
* **Stateless Protocol**:  
  Each request is independent; the server does not remember past requests.

**HTTP Request**

An HTTP request has two main parts:

1. **Headers**:
   * Contains metadata like:
     + Browser info
     + Cookies
     + OS details
     + Connection type
   * Includes **Request Method**.
2. **Body/Payload**:
   * Optional. Contains **data** sent to the server (e.g., form submissions, file uploads).

**HTTP Methods**

1. **GET**:
   * Purpose: Retrieve data from the server.
   * Example: Opening a webpage sends a GET request.  
     GET / HTTP/1.1
2. **POST**:
   * Purpose: **Create data** on the server.
   * Used when submitting forms, signing up, or adding new content.
   * Example: Submitting signup details to /signup.
3. **PUT**:
   * Purpose: **Create/Update data** at a *specific location*.
   * Key Difference with POST:
     + **POST**: Location is unknown; server determines the final resource location.
     + **PUT**: Client specifies the resource location.
     + PUT is **idempotent**: Sending the same PUT request multiple times has no side effects.
   * Example: Uploading a file to /files/abc.pdf.
4. **HEAD** (not discussed in depth):
   * Similar to GET but only retrieves headers, not the body.

**Example Request Flow:**

1. **Client Request**:  
   Browser sends a request to https://scaler.com (default port: 443).
   * If requesting /meetings/I/XYZ:
     + A GET request is sent to the path.
   * If signing up:
     + A POST request is sent to /signup with form data in the body.
2. **Server Response**:
   * Processes the request.
   * Fetches data from the database or backend logic.
   * Sends back an HTTP response with the required content.

**Key Notes:**

1. **Request Headers**: Include critical information like method, cookies, browser, etc.
2. **Request Payload**: Data sent with the request body (e.g., form values for signup).
3. **Statelessness**: HTTP is stateless, meaning the server doesn't retain previous request context.

**Summary of HTTP Methods:**

| **Method** | **Purpose** | **Example Use Case** |
| --- | --- | --- |
| GET | Fetch data | Opening a webpage |
| POST | Create new data | Submitting a signup form |
| PUT | Update or create data at a path | Uploading a file to a specific path |
| HEAD | Fetch headers (no body) | Checking headers of a resource |

## HTTP Methods and Search API Design

**Introduction**

* The discussion revolves around choosing the appropriate HTTP method (GET or POST) for implementing a search API, especially for use cases like e-commerce platforms (e.g., Flipkart, Myntra).

**Key Points on Search APIs**

**GET vs POST for Search APIs**

1. **GET Request:**
   * Clients use GET requests to *retrieve* data.
   * Suitable for searches when the query parameters (filters, sorting, etc.) are small.
   * **Limitation**: Query parameters have a character limit (e.g., ~1024–2048 characters depending on the server/browser).
2. **POST Request:**
   * **Used for search APIs with complex queries** that exceed GET's character limit.
   * POST has no formal size restriction on the payload.
   * Example:
     + A search query with multiple filters (tags, prices, brands, sorting, etc.) can become large.
     + Hence, many platforms (like Myntra) use POST for search operations.

**Why POST for Search?**

* While the intent of a search is to retrieve data (suggesting GET), POST is chosen **due to practical payload size constraints**.
* Filters, sorting criteria, and additional parameters can make the query exceed GET's limits.

**HTTP Methods Key Insights**

* HTTP methods (GET, POST, PUT, DELETE, etc.) define *what a request does*, but they do not impose restrictions on functionality.
  + For example: **You can send data with a GET request** or **retrieve data using a POST request**.
* HTTP methods are just **conventions**.

**Caching in GET Requests**

* GET requests are **cacheable**, meaning the client or intermediate proxies can store responses for reuse.
* Caching is not automatic and requires implementation or configurations (manual setup).

**Example Explanation**

* The teacher references real-world documentation of a search API (e.g., Myntra).
  + The search method uses a **POST request** because the payload is large.

**Takeaways**

1. **POST for search APIs**: When filters, sorting, and payloads grow large, POST avoids character limit issues.
2. **No restrictions**: HTTP methods are conventions, but servers ultimately decide what a method does.
3. **Caching**:
   * GET requests are easier to cache.
   * POST requests are typically not cacheable.

**Practical Suggestions**

* Students are encouraged to:
  + Experiment with web frameworks (e.g., Flask, Django, Spring Boot) to implement APIs.
  + Implement simple APIs to understand HTTP requests and responses.

## Key Topics Discussed

1. **FTP Protocol and Tokens**
   * FTP (File Transfer Protocol) does not use cookies.
   * Authentication or state management is handled within the FTP protocol (not external tokens or cookies like HTTP).
2. **HTTP Methods: PUT vs POST**
   * **POST**: Used to create new resources.
     + The server decides where to store the new resource.
   * **PUT**: Used to update/replace existing resources.
     + The client specifies the location where the resource should be stored.
   * Both **POST** and **PUT** can technically handle the same tasks, but PUT is often used for idempotent operations.
3. **Idempotency**
   * Idempotent requests: Multiple identical requests result in the same outcome (e.g., GET, PUT).
   * POST is **not idempotent** as it can create multiple new resources with repeated requests.
4. **Sockets, Ports, and Browser Behaviour**
   * Each **tab** in Google Chrome creates a **new process**.
   * Each **URL** connection in a tab uses a **new socket**.
   * A socket combines:
     + IP Address
     + Port Number
   * **Port Number**:
     + Used to identify processes on a server.
     + Example: HTTPS uses **default port 443**, HTTP uses **default port 80**.
   * Servers (e.g., Google) can handle multiple incoming connections (sockets) through **a single port**.
5. **How Requests are Handled**
   * The **Operating System** passes requests to the application without blocking.
   * The **Application Layer** maintains a **thread pool** to process requests concurrently.
6. **Example of Protocols Using Other Protocols**
   * Application Layer protocols can "wrap" other protocols:
     + **WebSockets**: Runs on top of **HTTP**.
     + **gRPC**: Internally uses HTTP/2.
7. **Connection Flow**
   * Both **sender's** and **receiver's port numbers** are sent in requests.
   * Multiple requests can come to the same port (e.g., 443), and the application processes them.
8. **Ports Usage Beyond HTTP/HTTPS**
   * A server might use multiple ports for:
     + **External communication** (e.g., HTTP/HTTPS).
     + **Internal communication** (e.g., connecting to databases or caches).
   * Port numbers like 3000, 6121, etc., can be used for other purposes.
9. **Port Differentiation in Servers**
   * Applications differentiate requests based on the:
     + IP Address
     + Port Number
     + Incoming Data
10. **Why HTTP/HTTPS Use Standard Ports**
    * Standard ports (443 for HTTPS, 80 for HTTP) simplify user experience.
    * No need to specify ports in URLs explicitly (e.g., https://example.com defaults to port 443).
11. **Request Execution Bottlenecks**

* Ports handle incoming requests very quickly.
* Actual request **processing** speed depends on the **CPU** and **application layer**, not the ports.

1. **One-to-Many Connections**

* A client connects to a server using a specific port (e.g., 443).
* The server can handle multiple client connections using the same port.

**Key Concepts Clarified**

* **Ports**: Identify processes on a machine.
* **Sockets**: Endpoint for communication between two machines.
* **Threads**: Used to process multiple requests in parallel at the application layer.
* **Standard Ports**:
  + HTTP: 80
  + HTTPS: 443
* **Application Layer Protocols**: Can build upon each other (e.g., WebSockets on HTTP).

**Simplified Examples**

1. **HTTP vs FTP**:
   * HTTP can use cookies; FTP does not.
2. **Sockets in Browsers**:
   * Each Chrome tab = New Process → New Socket for each connection.
3. **Server Ports**:
   * A single port (443) can handle thousands of connections simultaneously.