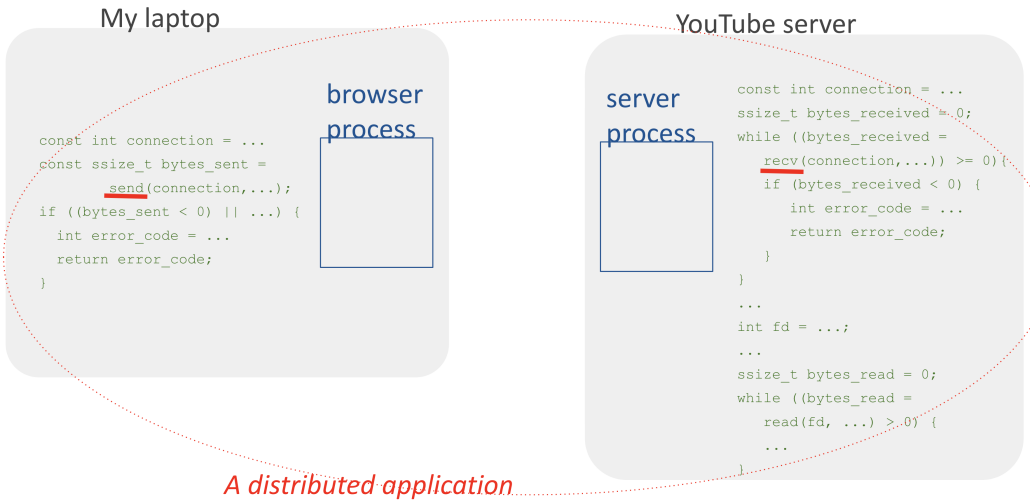
Chapter 1

Client/Server Model & The Web

Network ! Back to our Youtube example. (sorry for the delay was working on something else, I hope midterm went not too bad.)

Introduction

This chapter introduces the client/server model, a foundational concept of the modern web. We use YouTube as an example: when you play a video, your browser (client) communicates with a YouTube server to stream content. These two separate processes exchange messages over the network, forming a distributed application.



1.1 Distributed Applications

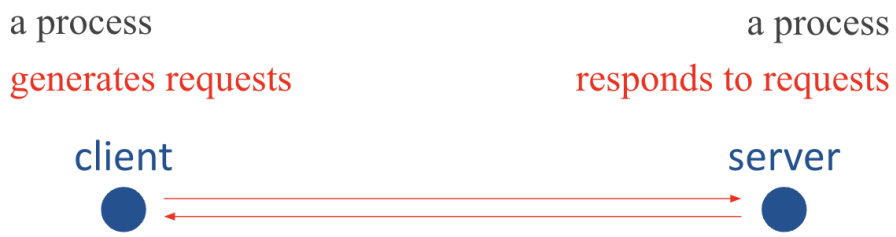
A **distributed application** consists of:

- Processes running on different computers,
- Exchanging messages over a network,
- Collaborating to achieve a shared goal.

1.2 Client/Server Architecture

Most distributed applications use a client/server architecture:

- One process acts as the **client**, making requests.
- Another acts as the **server**, responding to requests.

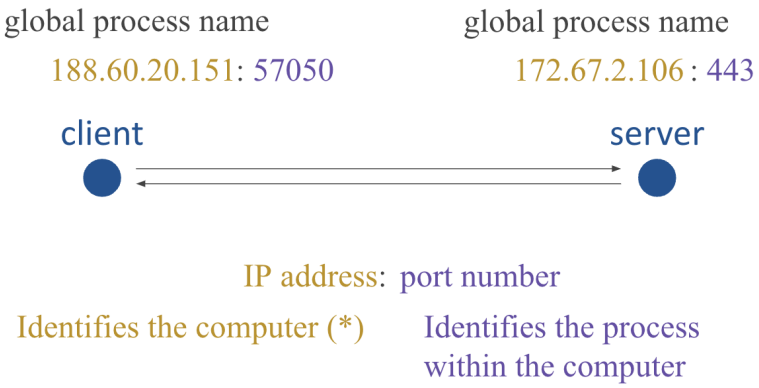


Clients and servers have clearly defined roles. Servers typically run on dedicated infrastructure, which today often means multiple data centers and many server processes.

1.2.1 Naming and Identifying Processes

To communicate, client and server processes need unique identifiers:

- **Local identifiers** (e.g., process ID or PID) only have meaning within a single computer.
- **Global identifiers** are needed for communication across computers.



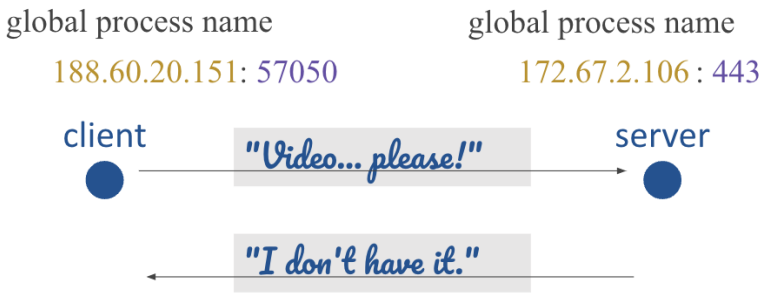
A global process name consists of:

- An **IP address** — uniquely identifies a computer (e.g., `172.67.2.106`),
- A **port number** — uniquely identifies a process on that computer (e.g., `443`, `57057`).

An IP address is like a street name; a port number is like a house number. Together, they uniquely identify a process on the network.

1.2.2 Discovering the Server Process

Before communication can begin, the client must discover the server's global name (IP address and port number). Once known, the client can initiate contact by introducing itself to the server.



Suppose the client requests a video from the server, but the server does not have the requested video. The server replies with a message indicating the resource is not found (for example, an HTTP 404 error).

If the server has the video, it responds with the requested data (such as an HTTP 200 OK response), and the client can begin receiving the video stream.

If the client does not follow the expected protocol—such as repeatedly sending requests for the same video without waiting for a response—the server may choose to ignore these requests or terminate the connection. This is a **protocol violation**.
Once the client knows the server's address, they can exchange messages according to a **communication protocol**.

1.2.3 Communication Protocols

A **communication protocol** defines the set of valid interactions between client and server processes. For example, a typical protocol might allow:

- The client requests information; the server responds with the requested data.
- The client requests information; the server responds that the information is unavailable.

If either party sends an unexpected or invalid message—such as the client repeating the same request rapidly—the protocol is violated. In such cases, the server may choose not to respond or may close the connection.

Analogy: Communication protocols in computers are similar to social protocols in conversation. If someone repeatedly asks questions without waiting for answers, the other person may stop responding.

Summary: Communication between distributed processes is only possible when the client can discover the server's address and both sides follow an agreed protocol. Violating the protocol typically ends the communication.

1.3 The HyperText Transfer Protocol (HTTP)

The HyperText Transfer Protocol (HTTP) is the foundation of communication on the World Wide Web. It defines how web clients (e.g., browsers) and web servers exchange information.

1.3.1 HTTP Requests and Responses

- **Request**: A web client sends an HTTP request to a server to retrieve or manipulate resources.
- **Response**: The web server processes the request and returns an HTTP response.
- **Common Request Methods**:
  - `GET`: Retrieves a specified resource.
  - `HEAD`: Retrieves metadata about a resource without the resource itself (e.g., object size or type).
  - `POST`: Sends data to the server, often used for form submissions.
- **Common Response Status Codes**:
  - `200 OK`: Request was successful.
  - `404 Not Found`: Requested resource could not be found.
  - `400 Bad Request`: Request was malformed.
  - `301 Moved Permanently`: Resource has been relocated to a new URL.

1.3.2 Web Objects

A web object is any resource accessible on the web, identified by a unique Uniform Resource Locator (URL).

- **Types**: Text files, images, videos, scripts, etc.
- **URL Example**: `https://actu.epfl.ch/image/142932/1920x1080.jpg`
- **Access**: Retrieved by a web client from a server using an HTTP request (e.g., `GET /image/142932/1920x1`
- **Uniqueness**: Each URL serves as a globally unique identifier for the resource.

1.3.3 Web Pages

A web page is a specific type of web object, typically composed of multiple resources.

- **Structure**: Consists of a base file (e.g., HTML) that defines the page's structure and references other objects.
- **Referenced Objects**: May include images, videos, scripts, or other web pages.
- **Example**: A web page might include an HTML file, CSS stylesheets, and embedded images, all fetched via separate HTTP requests.

1.3.4 Designing a Distributed Application

When developing a distributed application, it is essential to:

- Decide how many processes (and threads) are needed, and define the role of each.
- Design the communication protocol between the participating processes and threads.

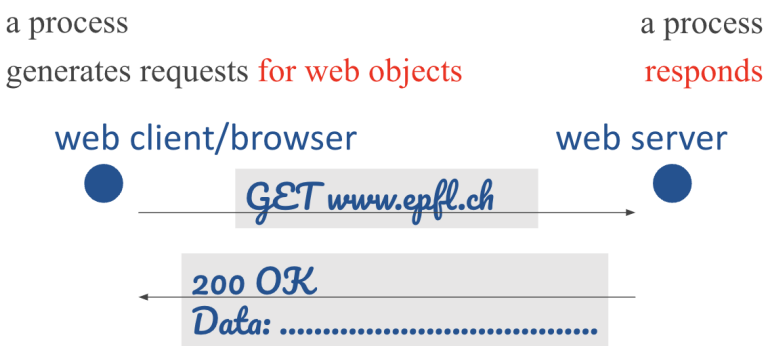
These decisions should be made before any coding begins.

A common and universal example of a client/server application is the web:

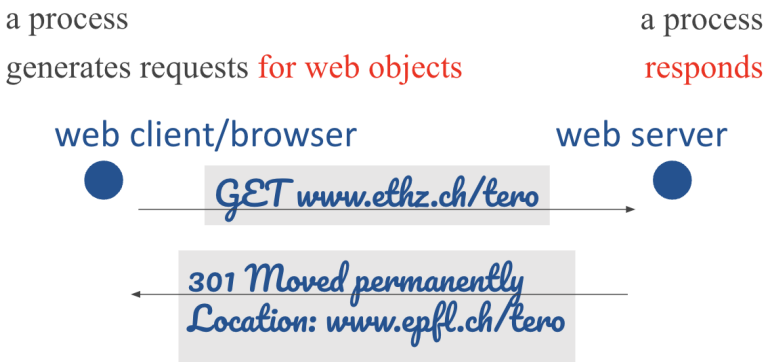
- **Web clients** (web browsers) generate requests for web resources.
- **Web servers** respond to these requests.

The communication protocol between a web client and a web server is the HyperText Transfer Protocol (**HTTP**), which is based on simple request and response interactions.

The most frequent request type is a `GET` request. The server can reply in several ways, each shown below:



When the server has the requested resource, it replies with a `200 OK` response, providing the object to the client.



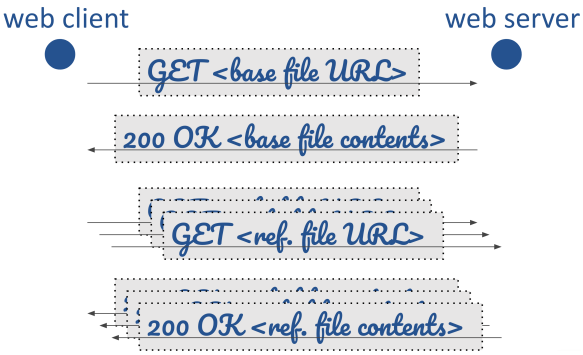
If the resource has been moved, the server replies with a `301 Moved Permanently` response, including the new location.

If the client's request is malformed or invalid, the server responds with a `400 Bad Request` error. This is rare in practice, because web browsers typically generate well-formed requests on behalf of users.

If the requested resource does not exist, the server sends a `404 Not Found` error.

Note: Most protocol errors are handled by the browser itself and are not directly seen by the user.

1.3.5 Example: Common Web Client/Server Exchanges



- A human user types a URL into their web client (e.g., a browser).
- The web client sends a `GET` request for the URL, which identifies the base file of a web page.
- The web server responds with a `200 OK` status, including the content of the requested base file.
- The web client parses the base file, discovers all additional URLs (e.g., images, scripts, stylesheets) needed to display the page, and sends a `GET` request for each one.

When you visit a web page, your web client sends multiple `GET` requests to the web server and receives multiple responses to fully render the page.

1.3.6 Stateless Protocols

- A server process does **not** maintain any information (or *state*) about previous interactions with a client.
- Here, *state* means data saved from past communication exchanges.
- By design, HTTP is a **stateless** protocol: each request from a client to a server is treated independently.

Let us contrast **stateless** versus **stateful** protocols:

- In a *stateful* interaction, the server remembers past conversations and can tailor responses accordingly.
- For example, in human conversation (like a classroom lecture), we recall previous topics discussed.
- In contrast, some services like a hospital system can be stateless: a new doctor might ask you to explain your medical history again because no information was saved.

If HTTP is stateless, then how do websites like Facebook or Amazon recognize you when you return, sometimes even before you log in?

1.3.7 Example: How Cookies Enable State

Suppose you open your web client and enter the URL `news.com/greece.html`.

- Your web client sends a `GET` request to `news.com` for the page `greece.html`.
- The web server responds with the page content and includes a **cookie** — a small piece of metadata that carries information about you or your preferences (in this case, that you are interested in Greece).
- Your web client stores this cookie.
- On subsequent requests to `news.com`, your client sends this cookie back to the server.
- The server reads the cookie, learns your interest in Greece, and can customize the response (e.g., show Greek recipes).
- The server can update or add new cookies to store additional inferred interests.

Over time, the server builds a profile about you based on cookies stored on your client, enabling a personalized experience despite HTTP being stateless.

1.4 Cookies

Cookies represent **state created by the web server but stored on the web client**. They serve as a mechanism to link multiple web requests from the same client, enabling session continuity despite HTTP's stateless nature. Through cookies, websites can recognize returning visitors and maintain user preferences across multiple visits.

1.4.1 Passing State to the Client

- Instead of storing client-specific information on the server, the server **passes** that state to the client.
- The client stores this state (in cookies) and sends it back on future requests.
- This approach reduces server memory and storage requirements.
- It also simplifies server design by offloading state management to the client.

Professor's Analogy (quoted): When I lived in Greece, clubs charged entrance fees but did not give out tickets. Instead, they stamped a client's forearm. If the client left and returned (e.g., to smoke), showing the stamp proved they had already paid.
Here, the club "passed the state to the client" by letting the client carry the proof themselves, rather than maintaining a list or checking IDs repeatedly. This simplified the job and reduced the club's need to keep state information.
Similarly, cookies pass state information from servers to clients, enabling a stateless HTTP protocol to behave like a stateful system.
Example of Cross-Site Information Transfer:
Continuing the previous example:

- **Initial Visit & Cookie Association**: A web client visits 'news.com'. The response from the 'news.com' server (or an embedded element therein) causes the client's browser to store a cookie that is associated with a different server, say 'cooking.com'.
- **Subsequent Visit to Different Server**: When the client later sends its first HTTP request to 'cooking.com', the browser automatically includes this specific cookie with the request.
- **Information Gained by Third Party**: As a result, the 'cooking.com' server receives this cookie. It can then learn information about the client (e.g., an inferred interest in "Greece," perhaps based on the context from 'news.com' or the cookie's content itself), even though this is the client's very first direct communication with 'cooking.com'.

1.4.2 Third-Party Cookies

A **third-party cookie** is a cookie set by a web server for a domain different from the one the user is currently visiting. This allows information about a user's browsing activity to be shared across different websites.

- **Definition**: A cookie created by one web server (e.g., an ad network) to be used when the client visits web pages hosted by other, different web servers.
- **Mechanism**:
    1. A user visits 'news.com'. The 'news.com' server's response might include instructions for the browser to fetch content from 'ads.com' (e.g., an image or script).

2. When the browser requests content from ‘ads.com‘, ‘ads.com‘ can set a cookie in the user’s browser. This is a third-party cookie from the perspective of ‘news.com‘.
3. Later, if the user visits ‘cooking.com‘, and ‘cooking.com‘ also embeds content from ‘ads.com‘, the browser will send the cookie previously set by ‘ads.com‘ along with the request to ‘ads.com‘.

- **Implication**: The third-party server (‘ads.com‘) can track the user’s visits across multiple sites (‘news.com‘, ‘cooking.com‘), building a profile of their interests even if the user has never directly visited the third-party’s website.

1.4.3 Cookie-less Tracking

Cookie-less tracking refers to techniques used by web servers to collect information about web clients and their activities without relying on traditional HTTP cookies.

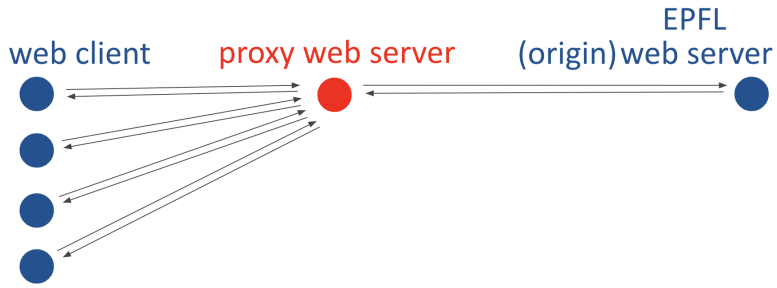
- **Methods**: These can include browser fingerprinting (collecting unique browser configurations), IP address tracking, ETags, or other header information.
- **Privacy Considerations**:
  - A common misconception is that cookie-less tracking is inherently more privacy-preserving simply because it avoids cookies.
  - **This is not necessarily true.** Cookies are just one mechanism for user profiling. The critical factor for privacy is the *nature and extent of the information collected and how it is used*, not the specific mechanism (cookies or otherwise).
  - Cookie-less *tracking* is still *tracking*. Its impact on privacy depends on what data is gathered and for what purpose.

1.5 Web Caching

Web caching is a core technique for improving the performance of web applications by storing copies of frequently accessed resources closer to the user.

1.5.1 Introduction to Web Caching

- A **web cache** (or **proxy web server**) is an intermediary server that stores (caches) copies of web content (e.g., HTML pages, images, files) served by origin web servers.
- It acts as a server to nearby web clients and as a client to the origin web servers.
- **Primary Goals**:
  - **Reduce Delay**: Clients experience faster load times as content is retrieved from a geographically closer cache rather than a distant origin server.
  - **Reduce Load on Origin Server**: Fewer requests reach the origin server, decreasing its workload and bandwidth consumption.



1.5.2 Web Caching Mechanism: An Example

Consider a web client accessing a resource (e.g., 'www.example.com/resource') from a distant origin server. A local proxy web server can be used:

1. **First Client Request (Cache Miss)**:
   - Client 1 sends a GET request for the URL to the nearby proxy server.
   - The proxy checks its local cache. If the resource is not found (a **cache miss**), the proxy forwards the GET request to the origin server.
   - The origin server responds to the proxy with the resource.
   - The proxy stores a copy of the resource in its cache and forwards the resource to Client 1.
   - Client 1 experiences a delay, potentially slightly longer due to the intermediary step.
2. **Subsequent Client Requests (Cache Hit)**:
   - Client 2 (or Client 1 again) requests the same URL from the proxy server.
   - The proxy checks its cache and finds a fresh copy of the resource (a **cache hit**).
   - The proxy immediately sends the cached resource to Client 2, without contacting the origin server.
   - Client 2 experiences significantly reduced delay.

1.5.3 Challenge: Ensuring Data Freshness

A critical challenge in web caching is ensuring that the cached data is not **stale** (i.e., an outdated version of the resource). Clients should receive the most current version.

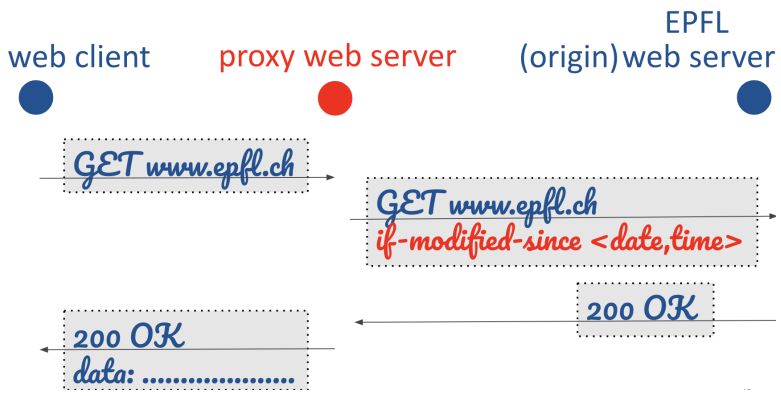
Cache Validation Mechanisms

To address stale data, caches use validation mechanisms:

- **Expiration Time / Max Caching Age**:
  - Origin servers can include HTTP headers in their responses to specify how long a resource can be considered fresh.
  - 'Expires': Provides a specific date/time after which the resource is stale.
  - 'Cache-Control: max-age=¡seconds¿': Specifies the maximum time in seconds that the resource can be cached without revalidation.
  - Once this period elapses, the cache considers its copy stale.
- **Conditional GET Requests**:
  - When a cached resource is stale (or if the cache wants to verify freshness), it can send a **conditional GET request** to the origin server.
  - This request asks the server to send the resource *only if it has been modified* since the version stored in the cache.
  - The 'If-Modified-Since' HTTP request header is used, containing the 'Last-Modified' timestamp of the cached version.

```
GET /resource.html HTTP/1.1
Host:  www.example.com
If-Modified-Since:  Wed, 21 Oct 2023 07:28:00 GMT
```

  - **Server Response**:
    - If the resource *has not been modified* since the specified date, the origin server responds with 'HTTP/1.1 304 Not Modified'. This response has no body, saving bandwidth. The cache can then serve its stored copy.
    - If the resource *has been modified*, the origin server responds with 'HTTP/1.1 200 OK' and the new version of the resource, which the cache then stores and forwards to the client.



Note: Web clients (browsers) also maintain their own local caches and employ similar mechanisms (expiration policies, conditional GETs) for resources they fetch.

1.5.4 Impact of Conditional GET on Performance

Does a conditional GET request significantly reduce delay?

- **Not always significantly for delay**: A conditional GET still requires a round-trip to the origin server to check for modifications. If the object is small, the time saved by not re-downloading it (in case of a '304 Not Modified') might be marginal compared to the round-trip time itself.
- **Significant for bandwidth and large objects**: If the requested object is large, receiving a '304 Not Modified' response avoids re-transmitting the entire object, leading to substantial savings in bandwidth and a noticeable reduction in delay compared to a full download.

1.5.5 General Principles of Caching

- Caching is a **universal technique** for improving performance in systems where data is accessed repeatedly.
- **Core Idea**: When one entity incurs the cost to fetch data, cache it locally (or closer to other potential consumers) so that subsequent requests for the same data can be served faster and with less resource consumption on the origin.
- **Primary Challenge**: Ensuring **data freshness** or **cache coherency**—that the cached data accurately reflects the current state of the origin data.
- **Common Solutions**:
  - Assigning an **expiration date** or **max caching age** to data.
  - Performing **dynamic checks** (e.g., conditional GETs) with the origin source to validate freshness. The utility of dynamic checks can depend on factors like data size and communication overhead.