Lecture X

# HTTP and CDNs

**CS 168, Fall 2024 @ UC Berkeley**

Slides credit: Sylvia Ratnasamy, Rob Shakir, Peyrin Kao,  Iuniana Oprescu

# HTTP Specification

Lecture X, CS 168, Fall 2024

# Brief History of HTTP

Development initiated by Tim Berners-Lee at CERN in 1989.

- 1991: Initial specification, HTTP/0.9, drafted.
- 1996: Standardized as HTTP/1.0.
- 1997: Updated to HTTP/1.1.
    - We'll use this version unless otherwise specified.

Driven by a need to share information between scientists.

- Needed a mechanism to transfer *hypertext* pages, with links to other pages.
- Resulting protocol: **H**yper**T**ext **T**ransfer **P**rotocol.

You can still view the first website ever made.

HTTP is a **client-server** protocol.

- One user is the client (e.g. your web browser, your terminal).
- One user is the server (e.g. the website).

HTTP **runs over TCP**.

- Client and server run a TCP handshake, and send data over the bytestream.
- No need to worry about packets being reordered, dropped, etc.
- Server listens for HTTP on well-known port 80. *(A later secure version uses 443.)*

HTTP is a **request-response** protocol.

- Client sends one request, and receives exactly one response.

## HTTP Requests

The request syntax is in human-readable plaintext (can be typed by a human).

**Version:** What HTTP version we're using.

**URL:** The *resource* we want to interact with.

- Intuition: The filepath of a file on some remote server.

**Method:** What we want to do with that resource.

- `GET`: Send me this resource. Originally, this was the only method.
- `POST`: Send data to the server (e.g. user submits a form).
- Other methods for *manipulating* content on the server, not just retrieving it:
  - `PUT`, `CONNECT`, `DELETE`, `OPTIONS`, `PATCH`, `TRACE`, etc.

| `GET` | `/projects/project1.html` | `HTTP/1.1` | `\r\n` |
|---|---|---|---|
| Method | URL | version | ends with a newline |

# HTTP Responses

**Version:** What HTTP version we're using.

**Status code:** A number, telling us what happened with the request.

**Description:** A description of the status code.

**Content:** The resource the user requested!

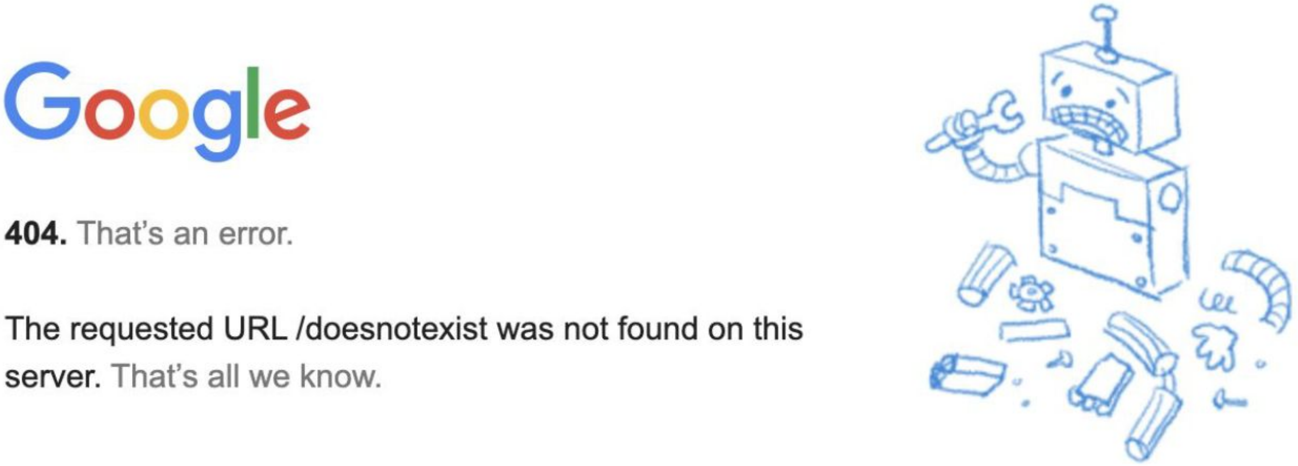
| `HTTP/1.1` | `200` | `OK` | `<html>Project 1 Spec...</html>` |
|---|---|---|---|
| version | status code | description | content |

## HTTP Responses – Status Codes

Status codes are used by the server to propagate information about the result of the request to the client.

Codes are classified into various categories, according to numeric value.

- 100s: Informational responses.
- 200s: Successful responses.
- 300s: Redirection messages.
- 400s: Client error.
- 500s: Server error.



# HTTP Responses – Status Codes

200s: Successful responses.

- **200 OK**: Request was successful.
  - Definition of success depends on the method in the request (e.g. GET, POST).
- **201 Created**: Request succeeded, and some new resource was created.
  - Seen generally in POST or PUT requests.

The HTTP status dog for
203 Non-Authoritative Information.

# HTTP Responses – Status Codes

300s: Redirection messages.

- Used when a server is telling a client they should go and look for the resource (specified by the URL) somewhere else.
- **301 Moved Permanently**.
- **302 Found**: Moved temporarily.
- Response contains extra context about where the resource moved.
  - `Location:` https://some.other.site/newpage.html
  - Encoded in a header (more on this soon).

Status codes let the client determine future behavior.

- Example: 301 means, always go to the new location.
- Example: 302 means, come back here to check again in the future.

# HTTP Responses – Status Codes

400s: Client error responses.

- **401 Unauthorized**: You need to authenticate (e.g. log in) to access this content.
- **403 Forbidden**: You are authenticated (server knows your identity), but access is still forbidden.
- **404 File Not Found**: You are requesting a file that doesn't exist.

500s: Server error responses.

- **500 Internal Server Error**: Server hit an error processing your request.
- **503 Service Unavailable**: Server cannot respond at the current time.

Status codes let the client determine future behavior.

- Example: 401 means, ask the user to log in.
- Example: 403 means, show an error message.

## HTTP Responses – Status Codes

Sometimes, which status code we should use is ambiguous.

Example: Request the Google homepage with HTTP/0.9.

- Maybe Google should respond with: **505 HTTP Version Not Supported**.
- But Google actually responds with: **400 Bad Request**.

Usually, the category of error is the most important.

- In the example: 400 or 500 = error.
- Goal is to elicit the correct behavior from the client.

# HTTP Headers

Requests and responses can contain additional metadata in the form of **headers**.

- Headers aren't mandatory (though server/client might expect a header and error).

Some headers are optional information.

- `User-Agent`: What program (e.g. Firefox, Chrome) the client is using.
- *Could* result in different processing of the request.

Some headers are critical information.

- `Content-Type`: File type of the response. (e.g. HTML, JPEG image, MP4 video...)

# HTTP Header Classes – Request

Headers can be classified into three types.

**Request** headers pass information about the client to the server.

- `Accept`: What file type the client is expecting in the response. Examples:
  - `Accept: text/html`
  - `Accept: application/json`
  - `Accept: image/*`
- `Host`: If a server is hosting multiple websites, identifies which website the client is aiming to access.
  - `Host: google.com:80`
- `Referer`: How the client triggered this request (e.g. clicking a link on Facebook).
- `User-Agent`: What program (e.g. Firefox, Chrome) the client is using.

"Referer" was misspelled in the original spec. Oops.

## HTTP Header Classes – Response and Representation

**Response** headers are in the response, but *not* directly related to the content.

- `Date`: When the server generated the response.
- `Location`: In 300 redirect responses, where the content moved to.
- `Server`: What software the server used to generate the response.

**Representation** headers are used in both requests and responses to describe how the content is represented.

- `Content-Type`: File type of the response. (e.g. HTML, JPEG image, MP4 video...)
- `Content-Encoding`: How the response is encoded into bits.
  - `Content-Encoding: gzip` says the contents were compressed.

# HTTP Examples

Lecture X, CS 168, Fall 2024

## HTTP in Terminal

```
$ telnet google.com 80

Trying 2607:f8b0:4005:802::200e...
Connected to google.com.
Escape character is '^]'.

GET / HTTP/1.1
User-Agent: robjs
```

HTTP is a text-based protocol, so we can connect to the Google server, type requests, and read responses, all in the terminal.

Using port 80 for HTTP.

Request: Get homepage, using HTTP version 1.1.

Adding a header to tell the server what type of client I'm using.

# HTTP in Terminal

```
$ telnet google.com 80

Trying 2607:f8b0:4005:802::200e...
Connected to google.com.
Escape character is '^]'.

GET / HTTP/1.1
User-Agent: robjs

HTTP/1.1 200 OK
Date: Sat, 16 Mar 2024 18:33:08 GMT
Content-Type: text/html; charset=ISO-8859-1

<!doctype html><html lang="en"><head><meta content="Search the
world's information, including webpages, images, videos and more.
Google has many special features to help you find exactly what you're
looking for." name="description">...
```

Response starts with status code: 200 OK.

Headers tell us the response date, file type (HTML), and encoding (e.g. ASCII).

The page we requested. (Would look nicer in a browser.)

# HTTP Examples

Note: We need a URL in all requests (even POST requests).

- The URL tells the server how to parse the information in the body of the request.
- Example: `POST /send-money` and `POST /request-money` do different things.

Note: The HTTP request can contain data.

- POST and PUT requests might contain data.
- GET requests probably don't contain data.

Note: This response doesn't have any content.

- The `Location` header redirects the user to the `success.html` page.

Request

```
POST /send-money HTTP/1.1
User-Agent: robjs

target=alice&amount=10
```

Response

```
HTTP/1.1 201 Created
Location: success.html
```

## HTTP Examples

The request has data (PUT a file on the server).

The response does not have data.

The `Content-Location` header says that the file we uploaded is stored at `newfile.html`.

Request

```
PUT /newfile.html HTTP/1.1
User-Agent: robjs

<p>Some File</p>
```

Response

```
HTTP/1.1 201 Created
Content-Location: newfile.html
```

# Speeding Up HTTP

Lecture X, CS 168, Fall 2024

# Multiple HTTP Requests

Loading a single website can require multiple HTTP requests.

- One request for the HTML (text/formatting) of the page.
- Separate requests for every picture.
- Separate requests for scripts to make the page interactive.

Naive approach: Separate TCP connection for each request.

- We have to do a 3-way handshake for every request.

# Multiple HTTP Requests – Pipelining

Smarter approach: Allow multiple requests to be **pipelined** over the same TCP connection.

- Trade-off: The server must maintain more open connections.

# HTTP Cache Types

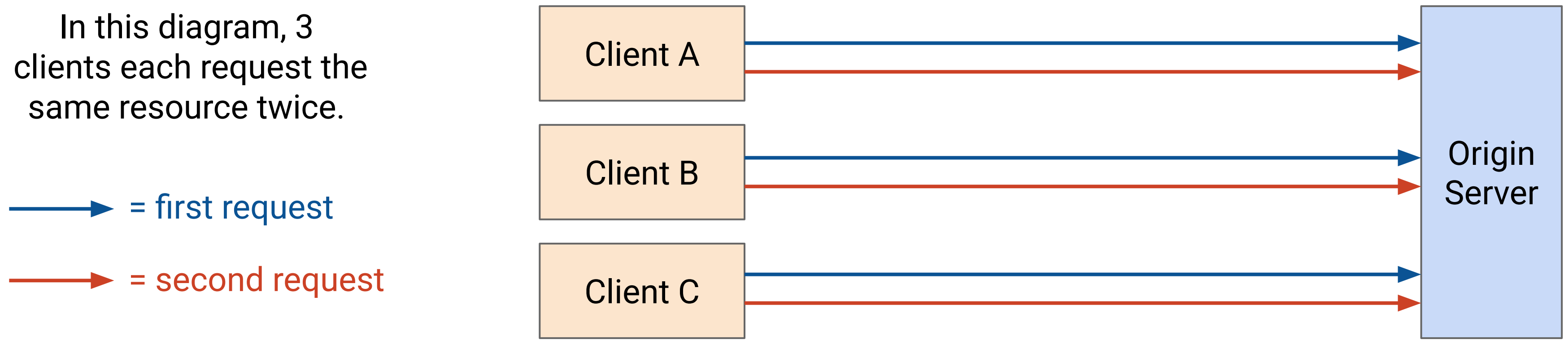
Optimization: Cache data to avoid sending duplicate requests for the same content.

Naive approach: Every request goes to to the origin server.

- Origin server: The server with the true (not cached) version of the content.

There are 3 types of caches: Private, Proxy, Managed.

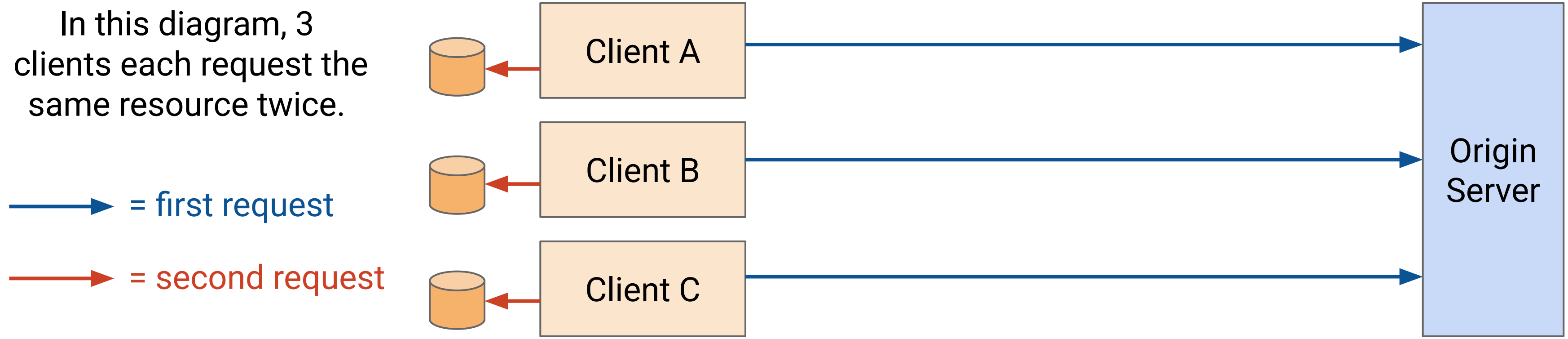
In this diagram, 3 clients each request the same resource twice.



# HTTP Cache Types

**Private** caches are tied to a specific end client (e.g. in a user's browser).

- When a user requests something for the second time, they can use the cache.



# HTTP Cache Types

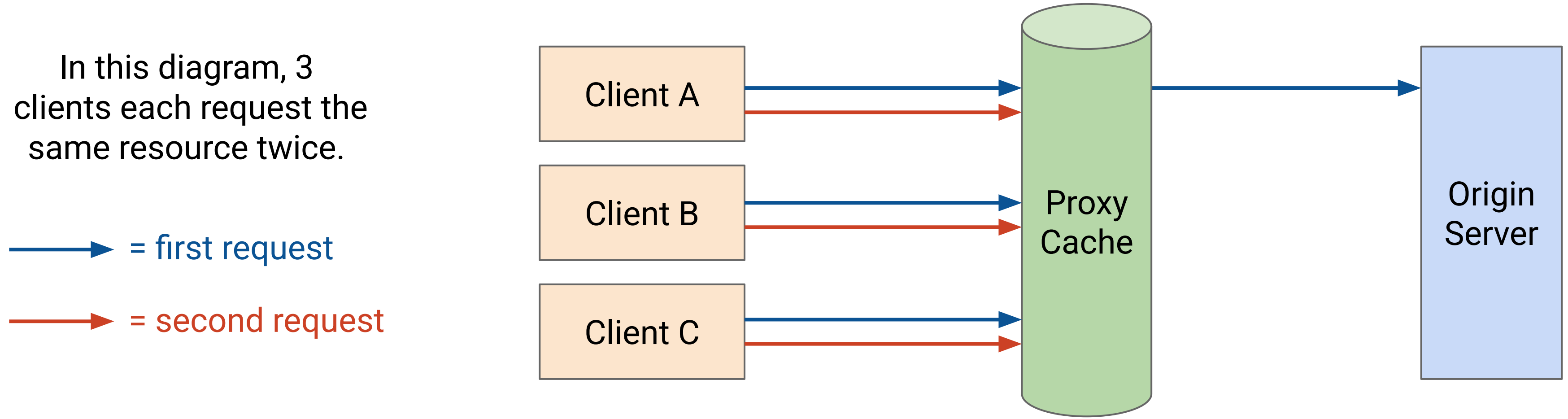
**Proxy** caches are in the network (not end host).

- The first client's first request goes to the origin server.
- All subsequent requests can be served by the proxy cache.

Proxy caches are operated by a third party (not the client or server).

Problem: Clients need to be redirected to the proxy cache somehow.

- DNS resolver could lie and say, "server's IP address is [*proxy cache address*]."



## HTTP Cache Types

**Managed** caches are in the network (just like proxy caches).

Managed caches are operated by the server (but cache server ≠ origin server).

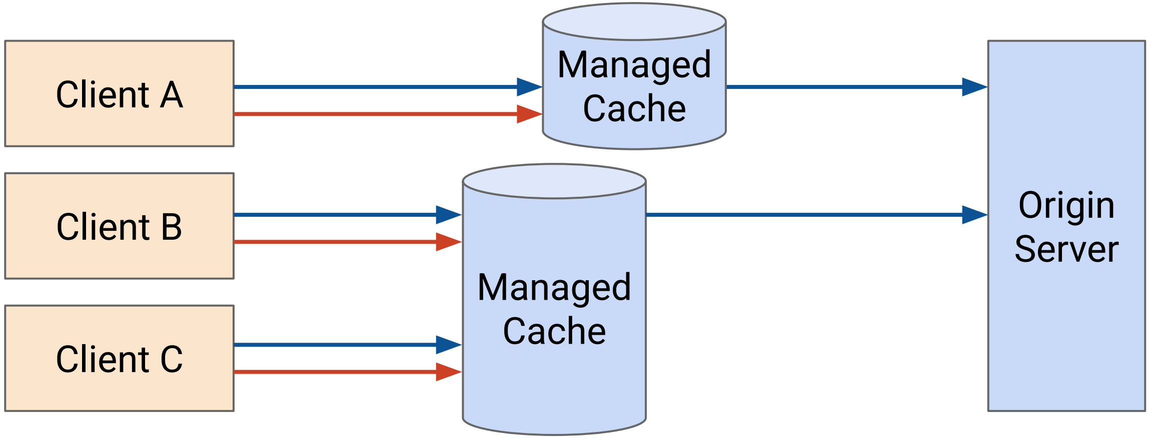
- Gives the server more control. Proxy cache might serve outdated/incorrect data.
- Server can redirect client to cache, because server knows about the caches.
    - In the HTML page: "Load the logo image from cache.google.com."

There can be multiple proxy/managed caches in the network.

In this diagram, 3 clients each request the same resource twice.

= first request

= second request



# Implementing Caching – Static vs. Dynamic Content

HTTP resources can be static or dynamic.

- **Static**: Stays the same for a long time. (e.g. Google logo image.)
- **Dynamic**: Generated on-demand for every request. (e.g. Search results.)

The server needs to tell everybody whether data can be cached, and if so, for how long.

- We can use HTTP headers!
- `Expires` header tells us when the content can be cached until.
  - Used in HTTP/1.0, obsoleted in HTTP/1.1.

Servers can't enforce that clients and caches actually obey the header.

- The header is more of a *request* to cache than a contract.

```
HTTP/1.0 200 OK

Date: Sat, 16 Mar 2024 19:40:24 GMT
Expires: Sun, 17 Mar 2024 19:40:24 GMT
```

The data in this response can be cached for 24 hours!

# Implementing Caching – Cache-Control Header

The `Cache-Control` header lets the server give more details on how to cache the data.

- `Cache-Control: private, max-age=86400`
  - Store in private caches for 24 hours.
  - Useful for responses that are different per user (private cache only).
- `Cache-Control: no-store`
  - This content cannot be cached.
- More complex policies are possible.
  - Example: Use the HEAD request method to re-request the header only and re-validate the data before using cache.

```
HTTP/1.1 200 OK

Date: Sat, 16 Mar 2024 19:40:24 GMT
Expires: Sun, 16 Mar 2024 19:40:24 GMT
Cache-Control: private, max-age=31536000
```

Servers could include both Expires (1.0) and Cache-Control (1.1) headers for compatibility.

1.1 client might ignore 1.0 header (and vice-versa).

## Benefits of Caching

Caching benefits everybody.

- Client can load pages faster.
  - With private caches, client saves time on subsequent accesses.
  - With proxy/managed caches, client gets data from a closer source.
  - Closer sources = lower RTT = higher TCP throughput.
- Less network bandwidth is needed.
  - Proxy caches are useful when there's low bandwidth out of a network.
- Reduces load on origin server.

Conveniently, the larger objects are *static*.

- Go to the origin server for dynamic content (e.g. small HTML page).
- Use a cache for static content (e.g. images, videos).

# Content Delivery Networks

Lecture X, CS 168, Fall 2024

HTTP

- Protocol Specification
- Examples
- Speeding Up HTTP

**Content Delivery Networks**

- **Deployment**
- Directing Clients to Caches

Newer HTTP Versions

# Using Caches for Content Delivery

How can application providers use caching to improve load time for users?

- Private caches are only useful if the same user accesses the same content repeatedly.
- Proxy caches aren't managed by the server.
  - Need some changes to tell the client about the caches (e.g. DNS trick).
  - Proxy cache might not obey the rules in the header.
- Managed caches are the best choice.
  - Controlled by the application provider.
  - Can be placed "close" to end users.
  - Application provider can redirect users to the caches.

## Content Delivery Networks

**Content Delivery Networks (CDNs)**: Deployments of servers that can serve content (HTTP resources).

Servers can be placed "close" to end users.

- Geographically.
- From a network perspective. (e.g. number of hops)

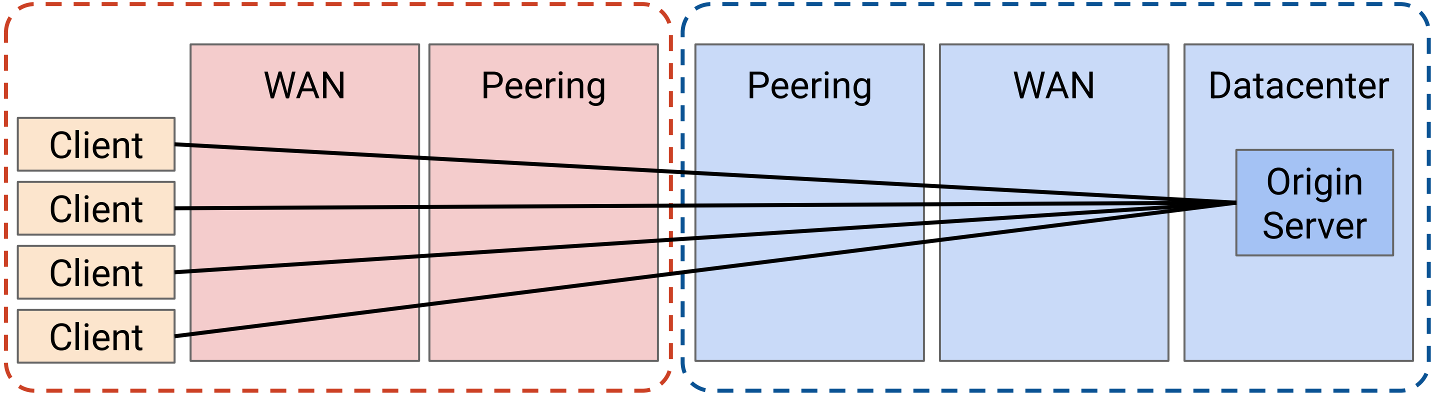
Benefits of CDNs:

- Higher performance for users.
  - Low-latency, high-throughput access to nearby server.
- Significant reductions in the bandwidth needed in the network.
- Reduces scaling needed for server infrastructure.
  - Adding more servers is easier than building one huge server.
- Provides better redundancy. If a server goes down, use another one.

# Deploying CDNs

Without CDNs, every request reaches the origin server.

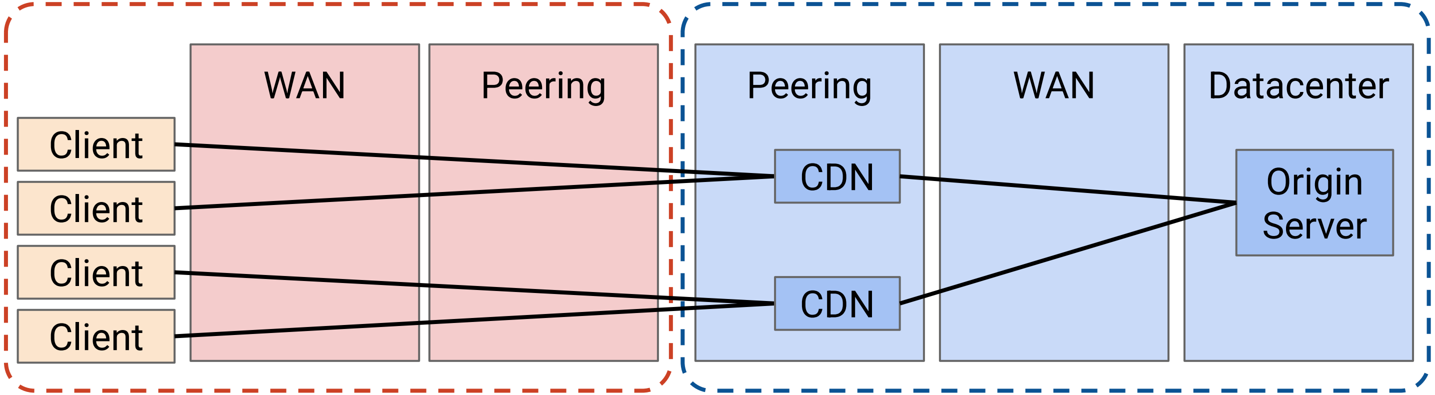
- Maximum latency → lowest performance.
- Maximum amount of "backbone" network traversed → highest cost to build.
- Scale must be supported on the origin server.



# Deploying CDNs

Application provider could deploy CDNs in its own network.

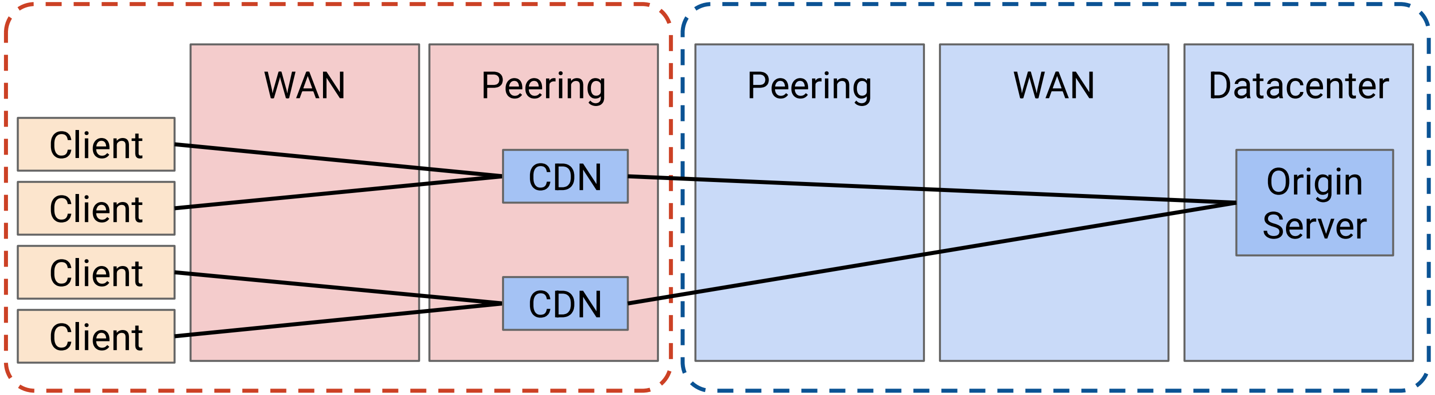
- Smaller servers at the "edge" of the application provider's networks.
- Reduces the volume of backbone traffic for the application provider.
- Reduces scale per deployment.



## Deploying CDNs

Caching can be pushed "deeper" into the network.

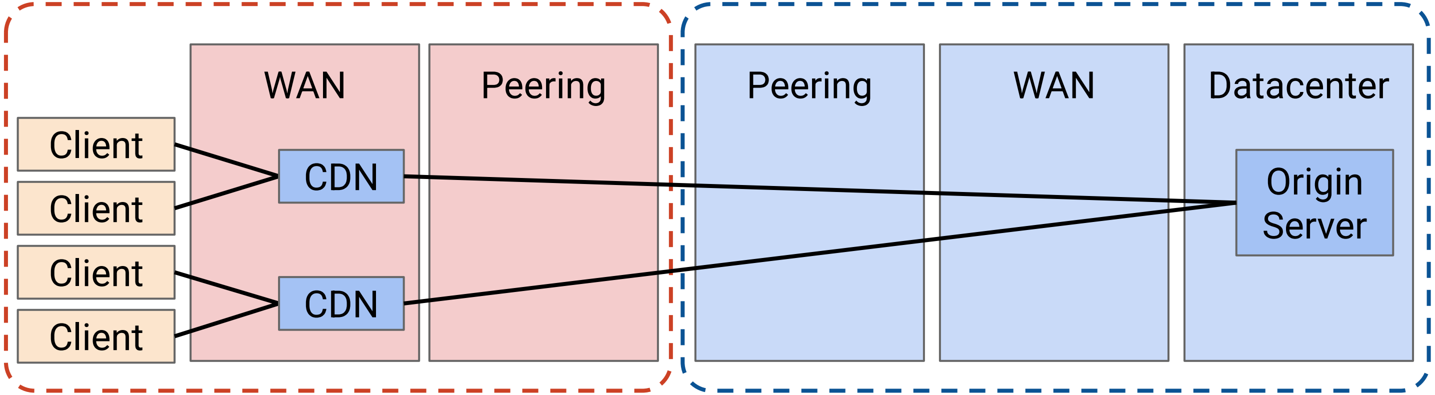
- Deploying CDNs in the ISP's network improves performance and reduces cost.
- ISP reduces their backbone network cost.
  - Less peering infrastructure needed.



# Deploying CDNs

Deployment depth is limited by efficiency.

- Need many users accessing the same content to justify the cost of the CDN.
- Cost-benefit trade-off between:
  - Cost of building a CDN.
  - Cost savings from building less network capacity.
- Example: Deploying a CDN in every user's home is probably not worth the cost.



## Large Global CDNs

Large application providers host their own CDNs.

- Netflix, Google, Amazon, Facebook.

CDN providers can host your service on their infrastructure for a fee.

- Cloudflare, Akamai, Edgio.
- Useful for smaller application providers.

Deployments either in their own networks, or directly into ISP networks.


## CDNs in ISP Networks

ISP companies often have their own content to serve.

- Example: Video-on-demand, or live TV, as part of TV+Internet bundles.
- CDN server infrastructure is also deployed by these ISPs.

Often a need for both third-party caches and the ISP's own infrastructure.

- Sandvine report (2023):
  - 15% of all Internet traffic is Netflix.
  - 11.4% is YouTube.
  - 4.5% is Disney+.
- Deploying caches can mean reducing ~25% of network capacity!

## Caching Server Deployments

CDN servers are highly optimized for content delivery and storage.

Flash appliance focus areas

- 2U for rack efficiency (no deeper than 29 inches)
- Enough low cost NAND to reach 24GB/s of throughput (<0.3 DWPD)
- Connect at up to 2X100G LAG
- 2 and 4 post racking
- AC or DC power
- Single processor

Storage appliance focus areas

- Large storage capacity
- 2U for rack efficiency (no deeper than 29 inches)
- Enough low cost NAND to reach 10GB/s of throughput (<0.3 DWPD)
- Network flexibility to connect at 6x100 LAG or up to 2x100GE
- 2 and 4 post racking
- AC or DC power
- Single processor

Example of Netflix server specs (you don't need to understand these).

# CDN Commercial Model

CDNs are mutually beneficial!

- Content provider gets better performance.
  - Better performance = more customers for both application and ISP.
- ISP gets lower bandwidth costs.

Cooperative commercial model:

- Content provider provides the servers for free.
- ISP hosts the servers for free.

In some cases, commercial negotiations are required.

- ISP and provider costs might not be equal as we get "deeper" into the network.

Becomes more difficult as there are more caching providers.

## Commercial Challenges – Fragmentation

Cache deployment makes sense if there are small numbers of large content providers.

There's a long tail of smaller content providers.

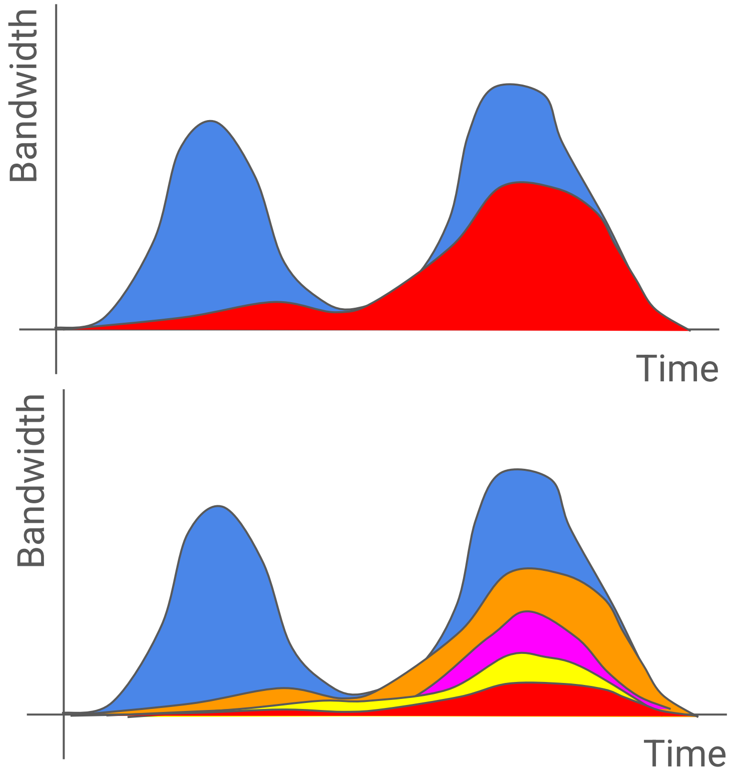
- Sandvine (2023): 4.5% is Disney+, 2.8% is Amazon Prime.

Idea: Can we have shared caching infrastructure?

- CDN InterConnect (CDNI) – IETF
- OpenCaching

Shared infrastructure is challenging!

- Who ensures quality?
- How are resources shared?



# Directing Clients to Caches

Lecture X, CS 168, Fall 2024

# Directing Clients to Caches

Recall our CDN model:

- Go to the origin server for dynamic content (e.g. small HTML page).
- Origin server directs the user to a cache for static content (e.g. images, videos).

If there are many CDN servers, which one should the server direct the user to?

Three approaches:

- Anycast.
- DNS-based load balancing.
- Application-level mapping.

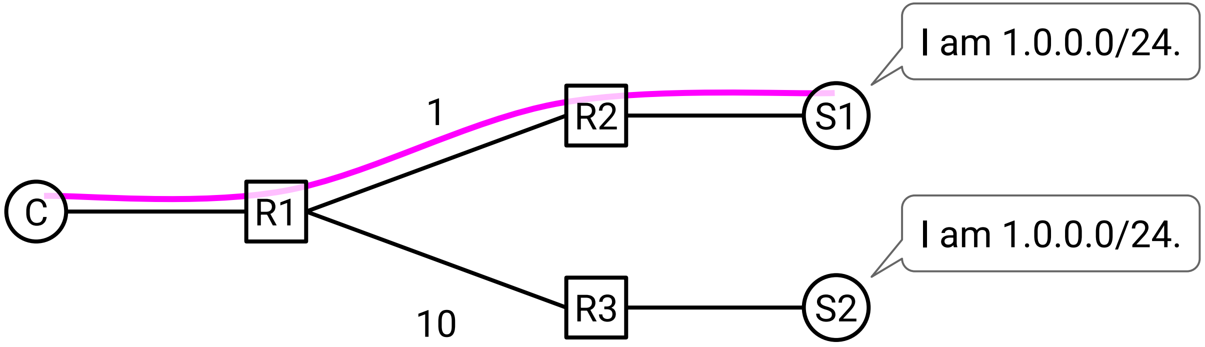
# Directing Clients to Caches – Anycast

Recall **anycast**: Advertise the same IP prefix from multiple locations.

- Least-cost routing chooses the best server to use.

Problem: Routing can change in the middle of a long-lived connection.

- Client starts a TCP connection with S1.
- Network topology changes. Client's packets are sent to S2 now.
- Server B doesn't have a TCP connection open and gets confused.
- Key issue: Routing thinks S1 and S2 are the same (since they have the same IP.)



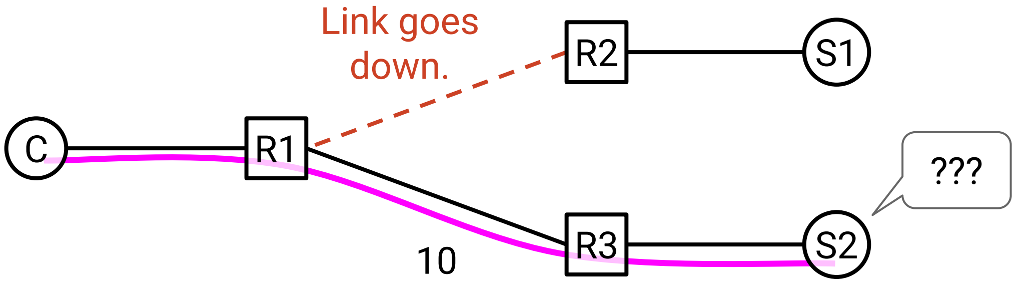
# Directing Clients to Caches – Anycast

Recall anycast: Advertise the same IP prefix from multiple locations.

- Least-cost routing chooses the best server to use.

Problem: Routing can change in the middle of a long-lived connection.

- Client starts a TCP connection with S1.
- Network topology changes. Client's packets are sent to S2 now.
- Server B doesn't have a TCP connection open and gets confused.
- Key issue: Routing thinks S1 and S2 are the same (since they have the same IP).



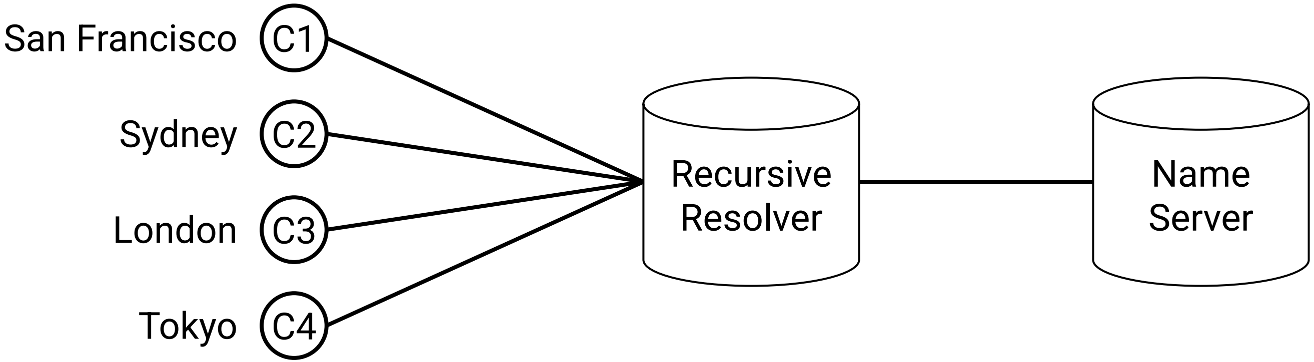
# Directing Clients to Caches – DNS-Based Load Balancing

Recall DNS-based load balancing: Map the same domain to different IP addresses, depending on where the query came from.

- Solves the anycast problem: Different servers have different addresses.

Problem: Granularity.

- If many users query through the same resolver, they all get the same address.
- Extensions are required to add end user information in the query.



# Directing Clients to Caches – Application-Level Mapping

Application directs the user to the cache.

- In the HTML page: "Load the video from sfo-cache.google.com."
- In the HTML page: "Load the video from mia-cache.google.com."

Benefits:

- Application knows the end user's address. (Solves the DNS granularity problem.)
- Allows for mapping at per-content item granularity.
  - Popular cat videos are served from many servers.
  - Obscure videos are served from fewer servers, closer to the origin.

Drawbacks:

- Still need to figure out which cache is "closest" to the client.
- Still need to decide the right strategy for failures (e.g. if a server goes down).

# Newer HTTP Versions

Lecture X, CS 168, Fall 2024

# HTTPS – Secure HTTP

Lots of applications run over HTTP.

- HTTP is very flexible. It can serve videos, webpages, etc.

As HTTP got popular, security became a concern.

- Network attackers (e.g. a malicious router) can read HTTP content.

HTTPS is a secure version of the protocol.

- Same syntax and semantics, but runs over TLS-over-TCP, instead of just TCP.
- TLS: Client and server exchange secret keys and encrypt their messages.
- Majority of traffic on the Internet (85.4% of websites) are now default HTTPS.
    - 85.4% of websites, according to W3Techs.
- HTTPS listens over port 443 (instead of 80).

# HTTP/2.0

HTTP/2.0 was introduced in 2015. (First new revision since 1997!)

Aims to decrease latency and improve page load speed.

- Data compression of headers.
- Server-side pushing: Servers can preemptively send data, without waiting for a request.
  - HTTP/2.0 is no longer a strict request-response protocol.
- Prioritization of requests.
- Better multiplexing of simultaneous requests.
  - Example: Ensure a small response doesn't get stuck behind a large one.

Widely adopted across client software (e.g. browsers) and CDNs.

# HTTP/3.0

HTTP/3.0 was introduced in 2022. (Not long after the previous update!)

Semantics are the same as HTTP/2.0, but runs over QUIC instead of TCP.

- QUIC = Quick UDP Connections.
    - Designed at Google. Standardized in IETF.
- QUIC is custom-built to work well with HTTP.
    - Abandons a classic network paradigm (layering).
    - In exchange, we get to simultaneously optimize Layer 7 and Layer 4 together.

# Summary: HTTP and CDNs

- HTTP is a protocol used to transfer data between a client and server.
  - Originally designed for HTML web pages.
- HTTP consists of request and response messages with headers in them.
  - Allows for different types of content to be carried over it.
- Performance of HTTP can be improved through caching static content.
  - HTTP provides means to control how this caching is used.
- Content Delivery Networks (CDNs) provide infrastructure to allow for this caching to be implemented to improve application performance.