2- Application Level Protocols HTTP 0.9/1.0/1.1/2

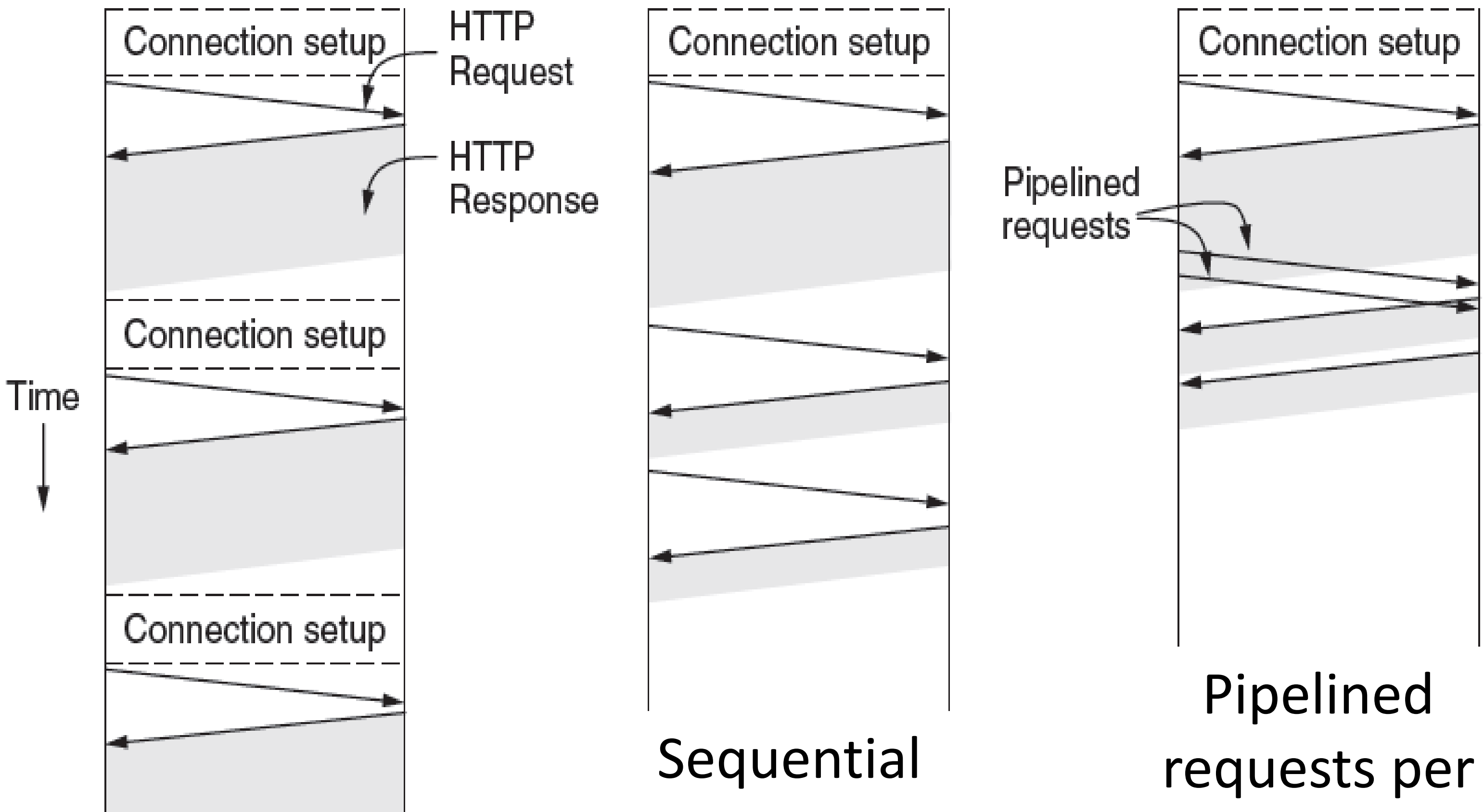
Part B

FROM LAST TIME

Review: Reducing Page Load Time

- Issue: A typical page is made up of many elements
  - Many elements may come from the same web server
- HTTP 0.9 required establishing a TCP connection per HTTP transfer
  - slow => do more than one HTTP transfer at a time
- HTTP 1.0 provides real headers but keeps TCP connection for framing HTTP requests
- HTTP 1.1 allows multiple HTTP requests to be sent sequentially over a single TCP connection

Can We Further Reduce PLT?



One request per connection

Sequential requests per connection

Pipelined requests per connection

Persistent Connections: Pipelining

- We would like to pipeline HTTP requests over a single TCP connection
  - Why isn't that done in HTTP 1.1?

- 19 years go by and Google wants better PLT
  - HTML/2!

- We get request pipelining and more

HTTP/2: RETHINKING EVERYTHING

HTTP/2 (2015)

- HTTP/2 evolved from Google SPDY, which started around2012
  - Standardization committee created HTTP/2
  - IETF RFC 7540, May 2015

- HTTP/2 preserves the semantics of HTTP 1.0 / 1.1
  - Client still says GET and server still responds OK

- However, the requests are
  - encoded differently (compressed)
  - transferred differently (streams and frames)

Issues

- We want pipelining!
  - HTTP/2 has pipelining
- HTTP header is encoded as text
- Headers have gotten very large
  - HTTP/2 compresses HTTP/1.1 headers
- Some elements on page are more important than others
  - HTTP/2 allows client to communicate “weights” with requests

Issues

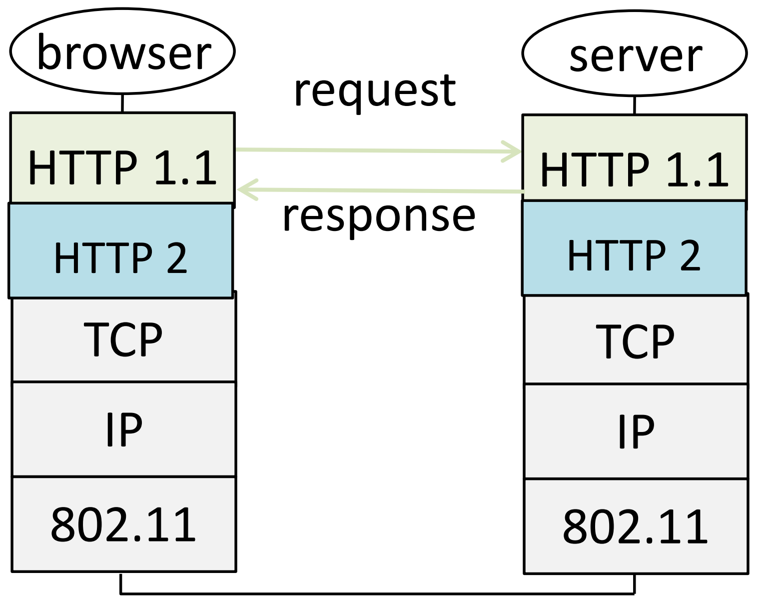
- Pipelining allows out of order replies by server
  - Server can apply it’s own weights to requests
    - (Neither client nor server has a complete view of how important something might be, or what it will cost to serve it)
- Client learns about embedded objects when it receives the page, but server knows about them already
  - “Server push” – here’s the response to a request you haven’t yet made

How It Fits Together

- Existing browser and web server software works with HTTP 1.1 headers
- Don’t want to rewrite/upgrade all that code
  - need to continue to speak HTTP/1.1 in any case
- Want to encode requests/response very differently, though

- Solution: Architect HTTP 2.0 so that:
  - it’s a transport for HTTP 1.1 messages
  - Using it could be implemented simply by writing a layer that packages an 1.1 message into HTTP 2.0 message

HTTP 2



This is the idea of how HTTP 2 fits in. A particular implementation might well combine HTTP 1.1 and HTTP 2

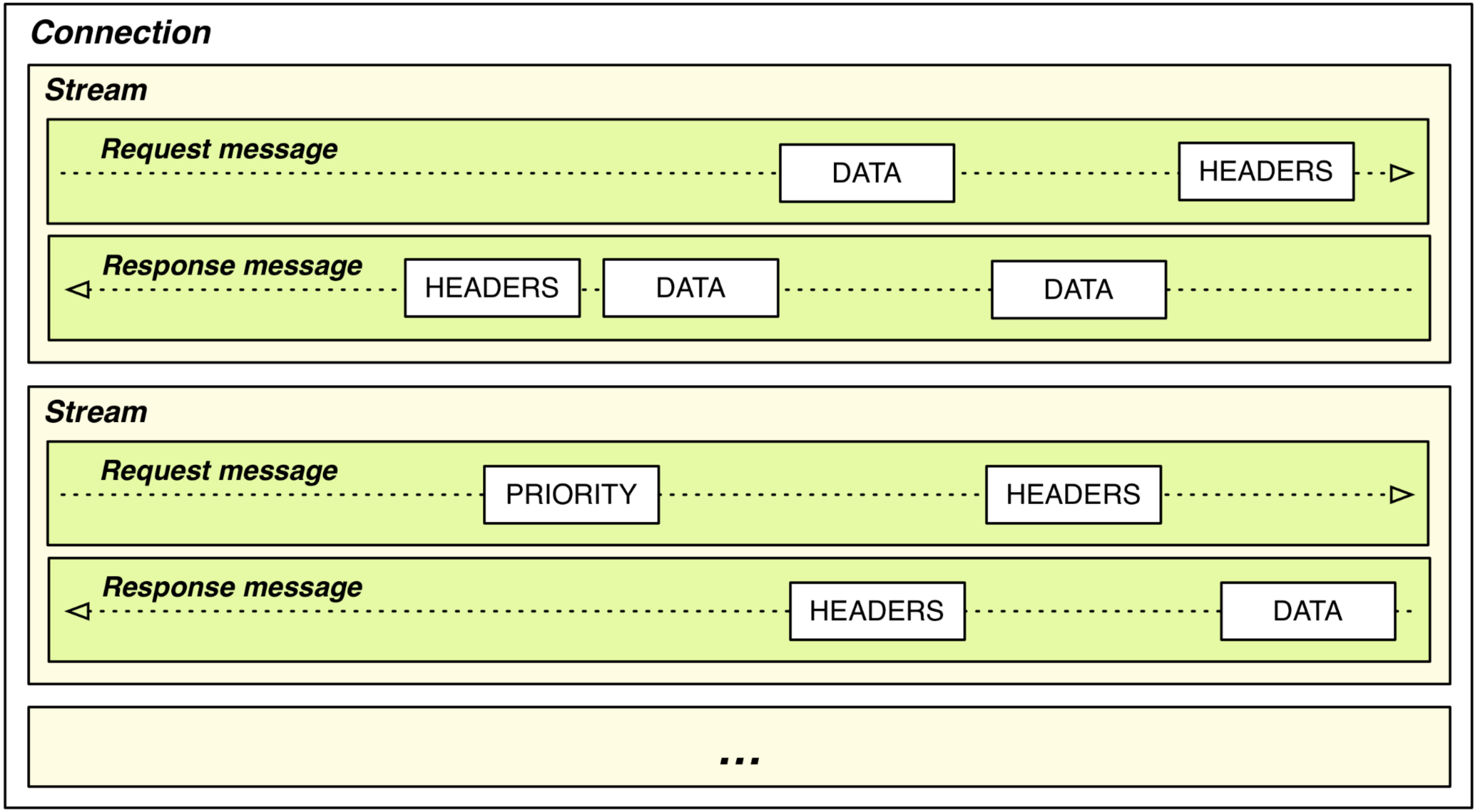
HTTP 2 – Main Features

- Allows “real pipelining” of requests on persistent connections
  - We have to “name” each request explicitly so that we can match responses to requests
    - Why can’t we use ordering of requests to match to responses?
- Compresses headers
  - Headers have gotten big
    - Cookies
- Servers can supply data that wasn’t requested
  - Called server push
  - “Here’s an image file needed by the HTML page you just fetched”
- Clients can advertise priorities among their requests
- “Real pipelining” allows servers to apply their own priorities, since they don’t have to reply in order

HTTP 2 – Streams and Frames

- An HTTP/2 *connection* is a TCP connection between client and server
  - long lived, just like HTTP 1.1
- An HTTP/2 *stream* is an ordered, bidirectional flow of information between client and server
- There is one connection between a client and server
- There is (roughly) one stream per HTTP request
- Multiple streams are being carried on the TCP connection at once

HTTP 2 – Streams & Frames



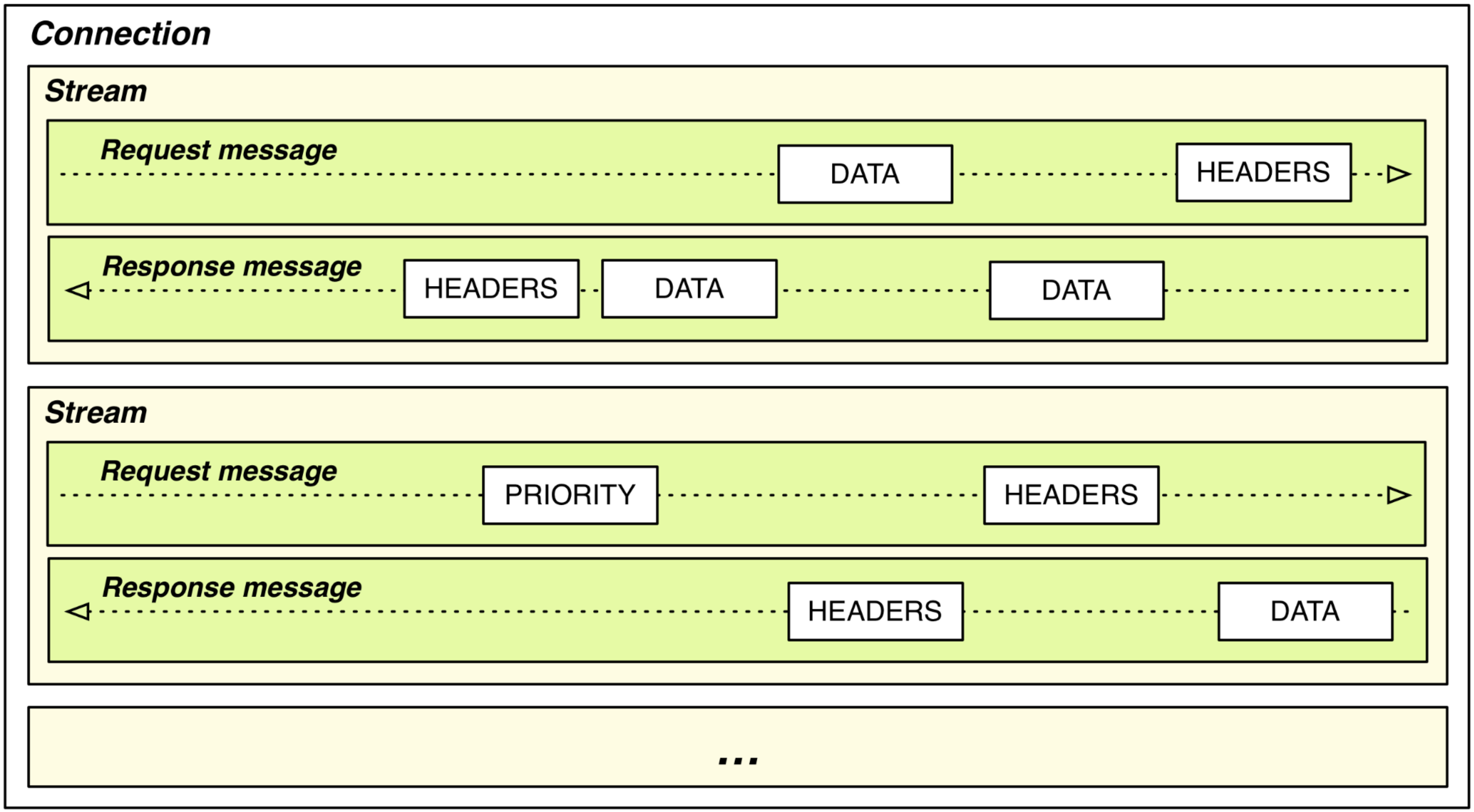
Streams

- Each stream has a unique ID
  - Successive stream IDs from one peer must be increasing
  - When run out of stream IDs, have to create a new connection
- A stream is created by sending a frame with a new stream ID
- Race condition if both ends try to create stream IDs
  - Client: “I choose 13” and Server: “I choose 13”
- Solution: statically partition possible names among possible name creators
  - in this case, “client” uses odd numbers, server uses evens
- *In general, what other solutions are there for choosing unique IDs?*

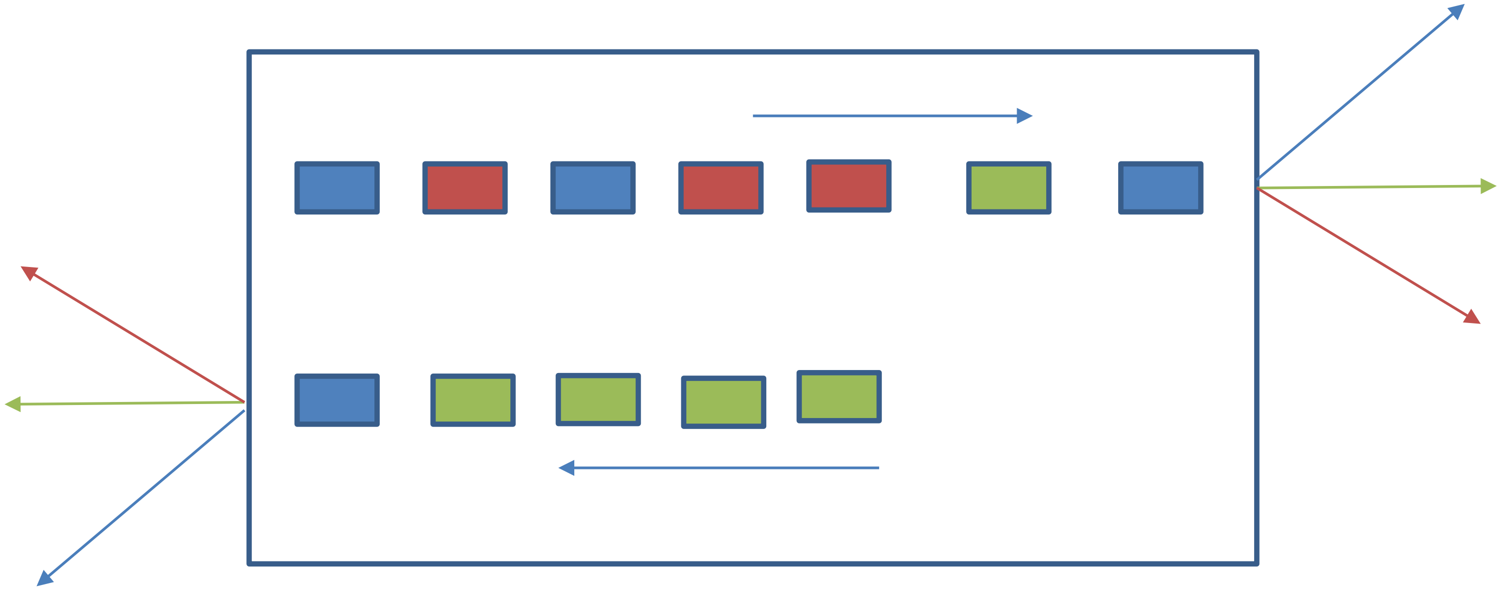
Frames

- An HTTP request is sent as a sequence of frames on a single stream
  - The response is sent as frames of the same stream in the opposite direction
- There are many streams using the TCP connection simultaneously
  - Many requests being conveyed in parallel
  - There is no particular ordering guarantees about delivery of frames in different streams
- An individual stream delivers its frames in order
  - Because TCP does

HTTP 2 – Streams & Frames

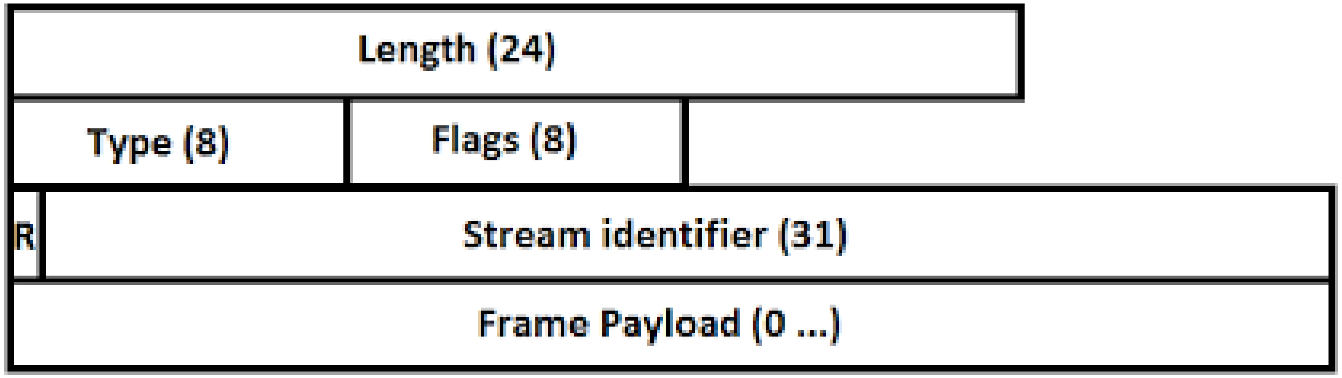


Viewed at the TCP level



Do frames need sequence numbers?

Frame Header



- Length: length of payload
  - header is always 9 bytes
- Type: frame type
- Flags: depends on type
- R: reserved; *"must be unset when sending and ignored when receiving"*
- Stream ID: 0x0 is reserved for frames associated with the connection (not an individual stream)

Frame Types

Frame type	Description
DATA	HTTP body
HEADERS	Header fields
PRIORITY	Sender-advised priority of stream
RST_STREAM	Signal termination of stream
SETTINGS	Configuration parameters for the connection
PUSH_PROMISE	Signal a promise (push) of referenced sources
PING	Measure roundtrip time and "liveness"
GOAWAY	Inform peer to stop creating streams for current connection
WINDOW_UPDATE	Connection flow control
CONTINUATION	Continue a segment of header block fragments

Simple encoding of an HTTP request

- Send a HEADER frame followed by zero or more CONTINUATION frames
  - Set END_HEADERS flag on last one
- Send DATA frames for request data, if needed
  - Set END_STREAM flag on last

- Response is the same, in reverse

HEADER frame

```
+---------------+
|Pad Length? (8)|
+-+-------------+-----------------------------------------------+
|E|                 Stream Dependency? (31)                     |
+-+-------------+-----------------------------------------------+
|  Weight? (8)  |
+-+-------------+-----------------------------------------------+
|                   Header Block Fragment (*)                 ...
+---------------------------------------------------------------+
|                           Padding (*)                       ...
+---------------------------------------------------------------+
```

Figure 7: HEADERS Frame Payload

- Padding is for security – obfuscate lengths
- Stream dependency – make this stream a child of named stream
    - If server can't make progress on parent, assign resources proportional to weights to children
- Header block fragment – take the HTTP 1.1 header and compress it, then send it in chunks (if necessary)
- Frame header flags: END_HEADERS and END_STREAM

DATA Frame

```
+---------------+
|Pad Length? (8)|
+---------------+-----------------------------------------------+
|                            Data (*)                         ...
+---------------------------------------------------------------+
|                           Padding (*)                       ...
+---------------------------------------------------------------+
```

Figure 6: DATA Frame Payload

PRIORITY Frame

```
+-+-------------------------------------------------------------+
|E|                  Stream Dependency (31)                     |
+-+-------------+-----------------------------------------------+
|   Weight (8)  |
+-+-------------+
```

Figure 8: PRIORITY Frame Payload

- E: exclusive bit – inserts this stream as only child of parent stream, moving existing children to be children of this stream

RST_STREAM Frame

```
+---------------------------------------------------------------+
|                        Error Code (32)                        |
+---------------------------------------------------------------+
```

Figure 9: RST_STREAM Frame Payload

- Ends a stream
  - Why is this useful?
    - Also have END_STREAM flag bit...

GOAWAY Frame

```
+-+-------------------------------------------------------------+
|R|                  Last-Stream-ID (31)                        |
+-+-------------------------------------------------------------+
|                      Error Code (32)                          |
+---------------------------------------------------------------+
|                  Additional Debug Data (*)                    |
+---------------------------------------------------------------+
```

Figure 13: GOAWAY Payload Format

- Closes connection
- Provides largest id of any stream that the server may have acted on
  - Why?

PUSH_PROMISE Frame

```
+---------------+
|Pad Length? (8)|
+-+-------------+-----------------------------------------------+
|R|                  Promised Stream ID (31)                    |
+-+-----------------------------+-------------------------------+
|                   Header Block Fragment (*)                 ...
+---------------------------------------------------------------+
|                           Padding (*)                       ...
+---------------------------------------------------------------+
```

Figure 11: PUSH_PROMISE Payload Format

- Allows server to send something not yet asked for
  - E.g., a style sheet or a javascript program or an embedded image
- Acts like a HEADERS frame
  - Can have CONTINUATIONs following for more header

PING Frame

```
+---------------------------------------------------------------+
|                                                               |
|                      Opaque Data (64)                         |
|                                                               |
+---------------------------------------------------------------+
```

Figure 12: PING Payload Format

- Is other end still there?
  - Responds with PING with ACK flag bit set
- Measure latency to other end
  - PING frames have highest priority...

WINDOW_UPDATE Frame

```
+-+-------------------------------------------------------------+
|R|              Window Size Increment (31)                     |
+-+-------------------------------------------------------------+
```

Figure 14: WINDOW_UPDATE Payload Format

- TCP does flow control on entire connection
  - but need flow control on a per stream basis as well

Getting There From Here

- HTTP 2 is intended as an optimized transport of HTTP requests
  - Needs to be backward compatible with HTTP 1/1.1

- Main problem:
  - How to tell if client and server can both speak HTTP 2?
    - Client could try HTTP 2 and then revert to 1.1
    - Client could start with HTTP 1.1 then upgrade to 2

Dynamically Upgrading to HTTP 2

- Client:

```
GET / HTTP/1.1
Host: server.example.com
Connection: Upgrade, HTTP2-Settings
Upgrade: h2c
HTTP2-Settings: <base64url encoding of HTTP/2
SETTINGS payload>
```

Server Refuses Upgrade

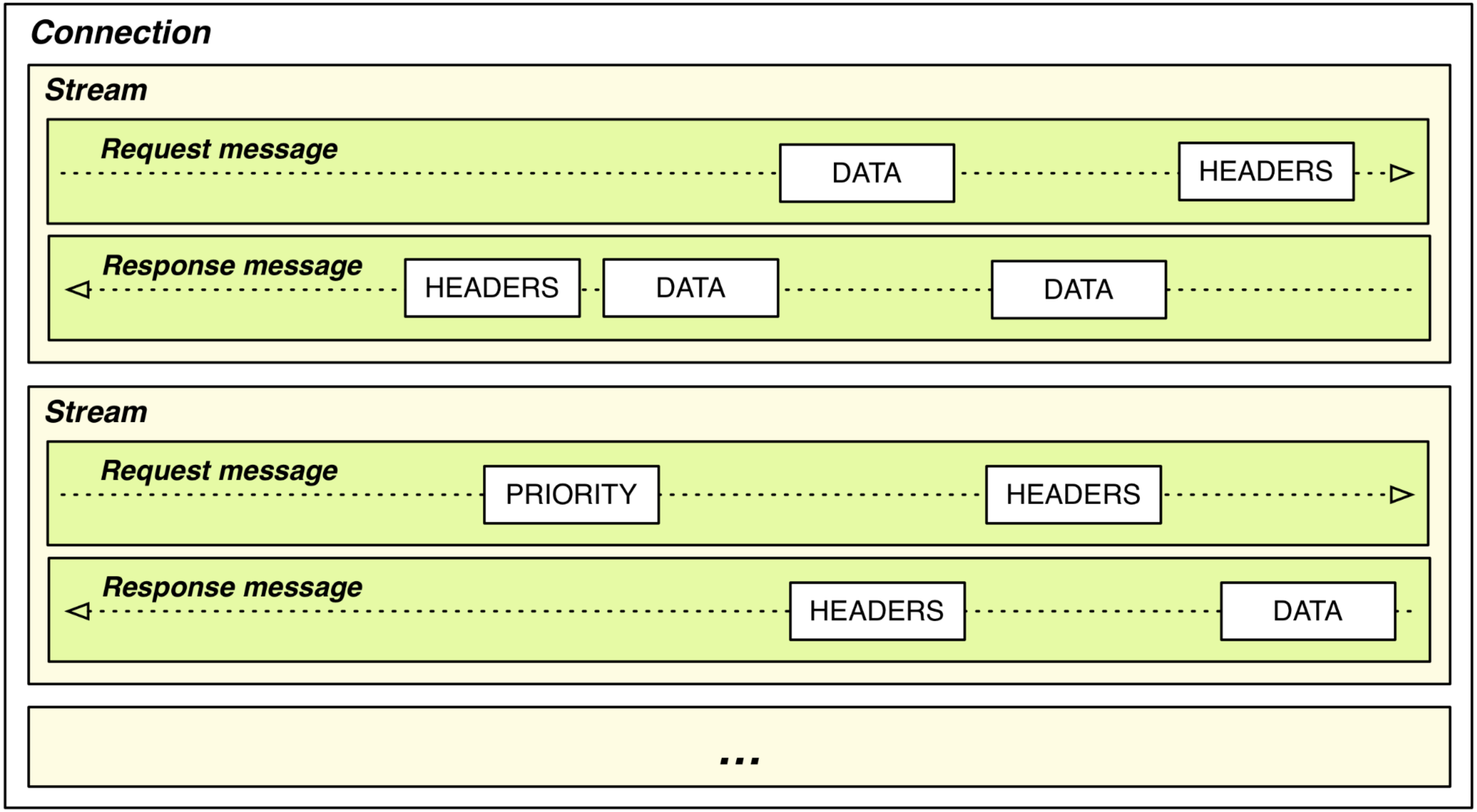
- Server may simply not recognize the upgrade request if it isn't HTTP 2 capable

```
HTTP/1.1 200 OK
Content-Length: 243
Content-Type: text/html

  ...
```

Server Wants to Upgrade

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: h2c

[ HTTP/2 connection ...
```

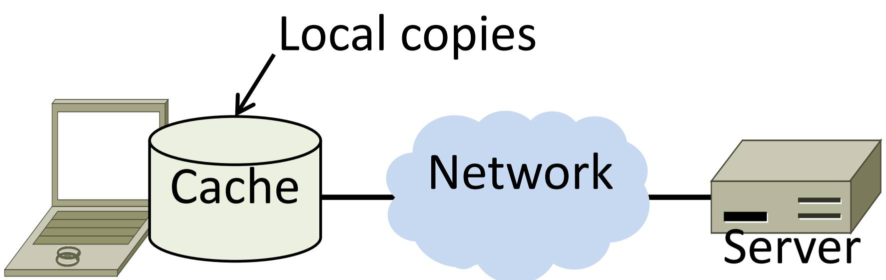
HTTP 2 Wrap-up



WWW PERFORMANCE: CACHING AND CDN’S

Web Caching

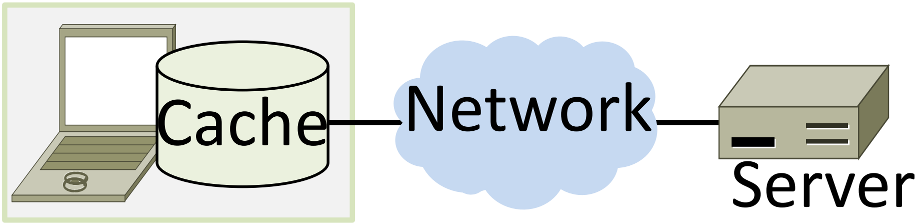
- Users often revisit web pages
  - Big win from reusing local copy!
  - This is caching



- Key question:
  - When is it OK to reuse local copy?

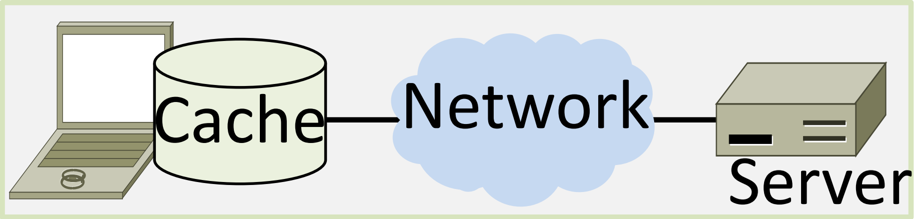
Web Caching (2)

- Locally determine copy is still valid
  - Based on expiry information such as “Expires” header from server
  - Or use a heuristic to guess (cacheable, freshly valid, not modified recently)
  - Content is then available right away



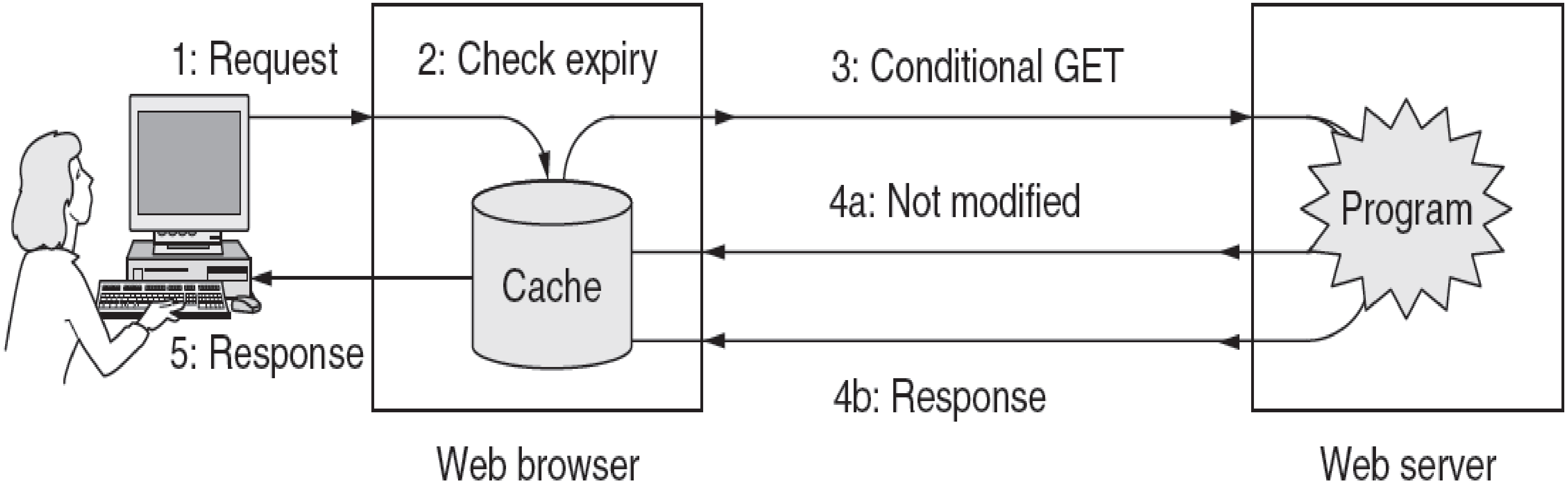
Web Caching (3)

- Revalidate copy with remote server
  - Based on timestamp of copy such as “Last-Modified” header from server
  - Or based on content of copy such as “Etag” server header
  - Content is available after 1 RTT



Web Caching (4)

- Putting the pieces together:



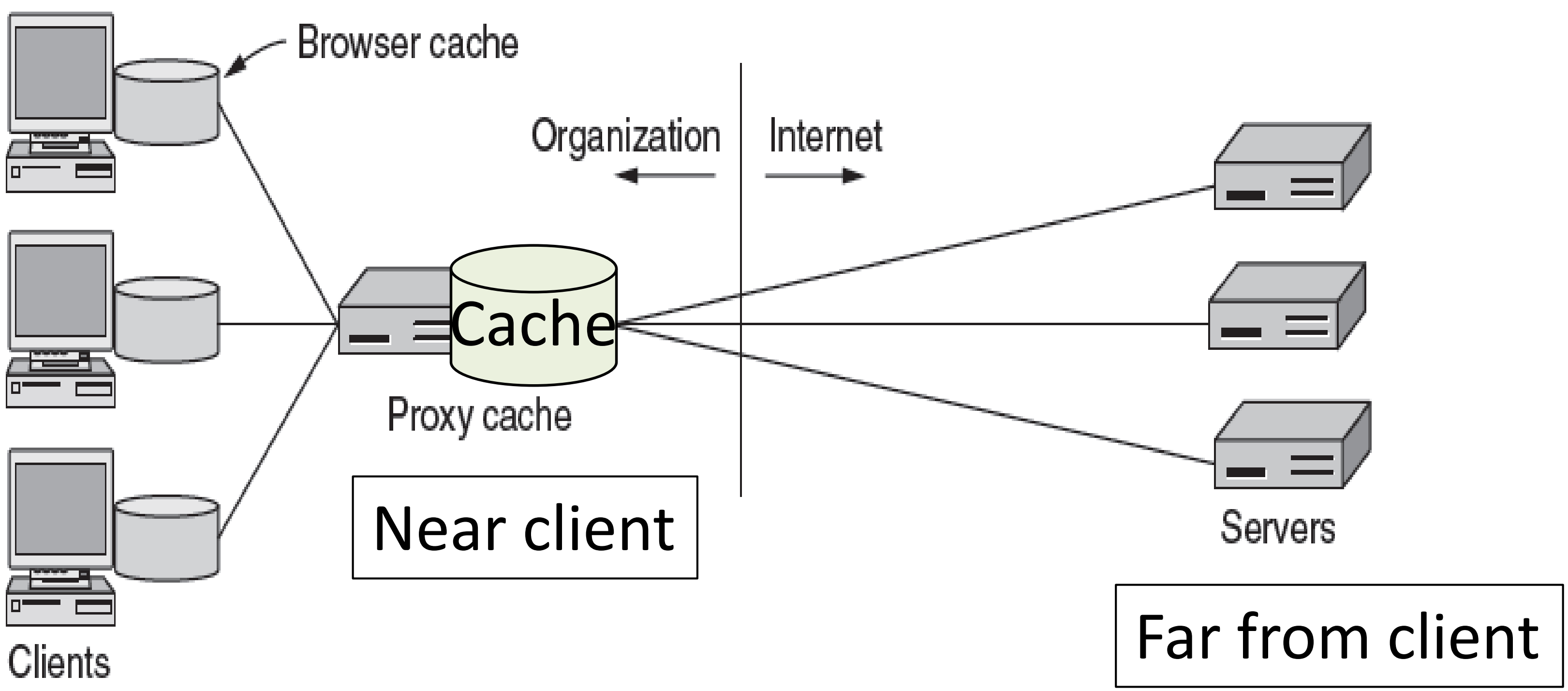
Web Proxies

- Place intermediary between pool of clients and external web servers
  - Benefits for clients include caching and security checking
  - Organizational access policies too!

- Proxy caching
  - Clients benefit from larger, shared cache
  - Benefits limited by secure / dynamic content, as well as "long tail" of page popularity distribution

Web Proxies

- Clients contact proxy; proxy contacts server

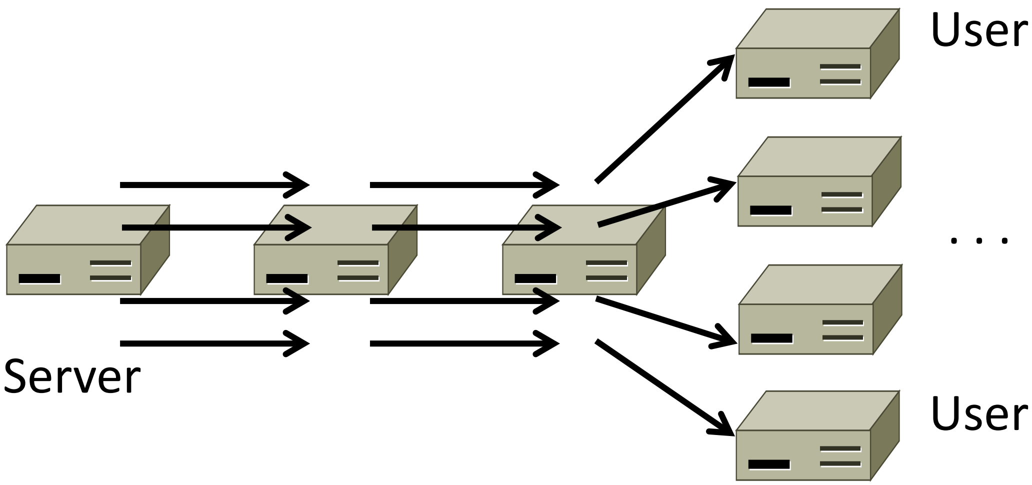


Content Delivery Networks

- As the web took off in the 90s, traffic volumes grew and grew. This:
  1. Concentrated load on popular servers
  2. Led to congested networks and need to provision more bandwidth
  3. Gave a poor user experience
- Idea:
  - Place popular content near clients
  - Helps with all three issues above

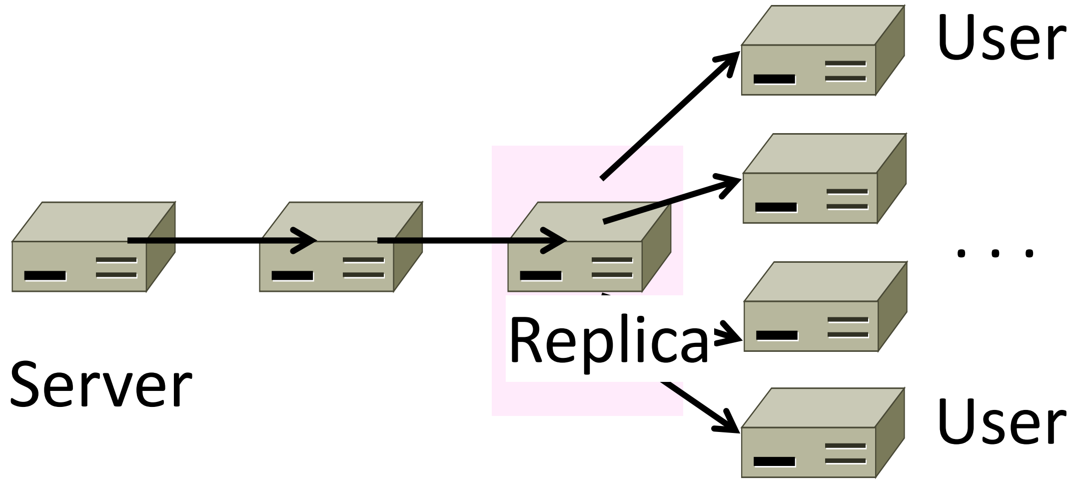
Before CDNs

- Sending content from the source to 4 users takes 4 x 3 = 12 “network hops” in the example



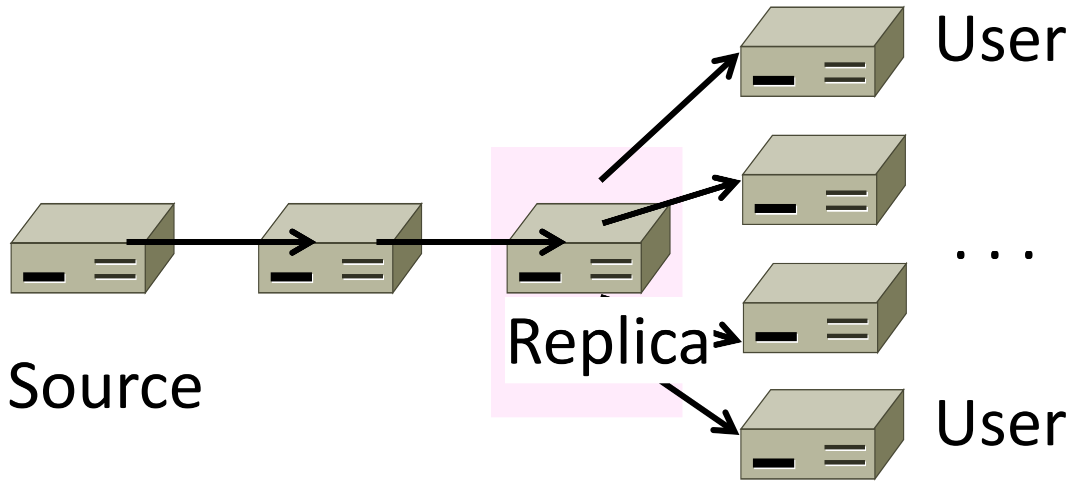
After CDNs

- Sending content via replicas takes only 4 + 2 = 6 “network hops”



After CDNs

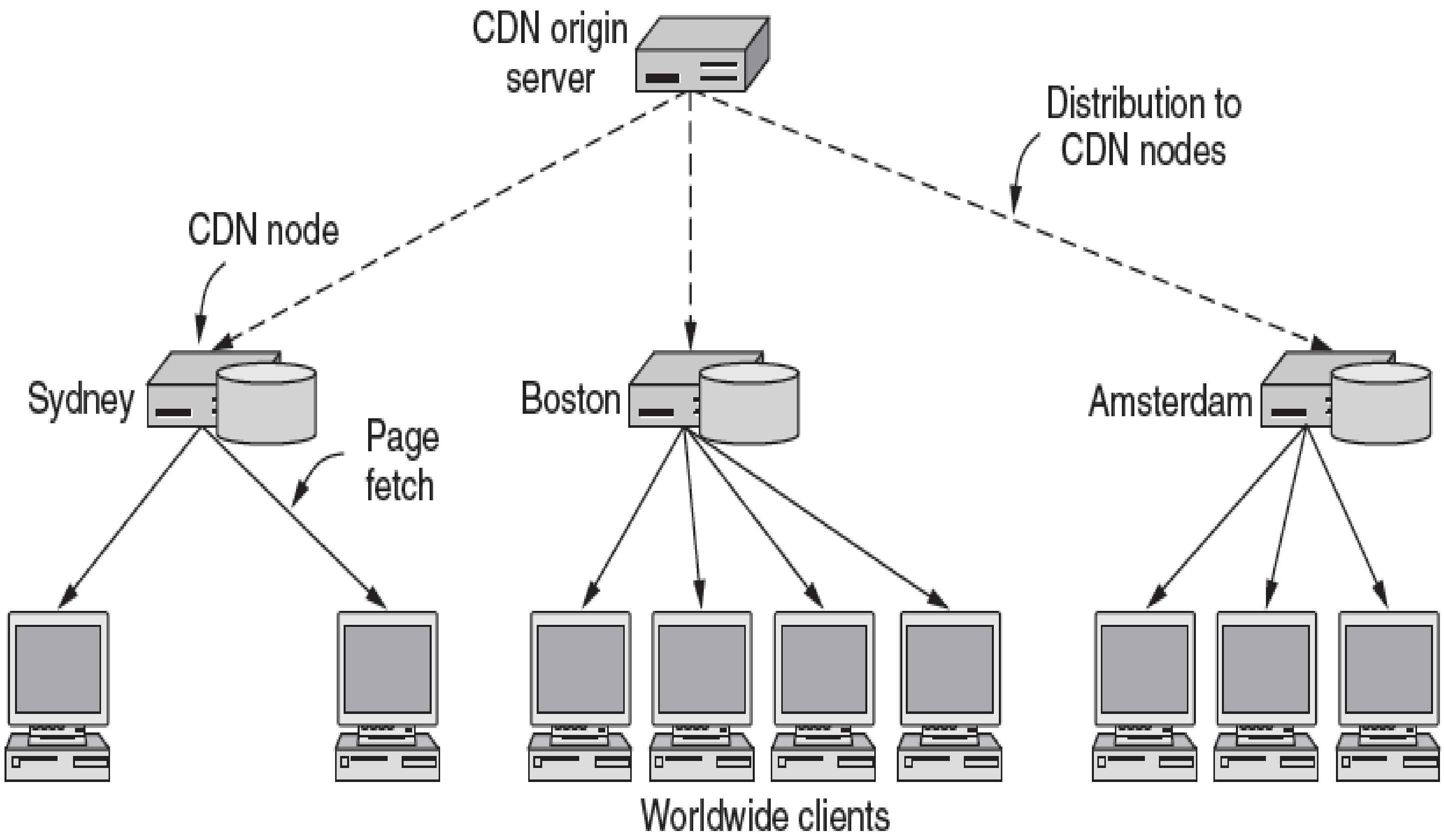
- Benefits assuming popular content:
  - Reduces server, network load
  - Improves user experience (PLT)



How to place content near clients?

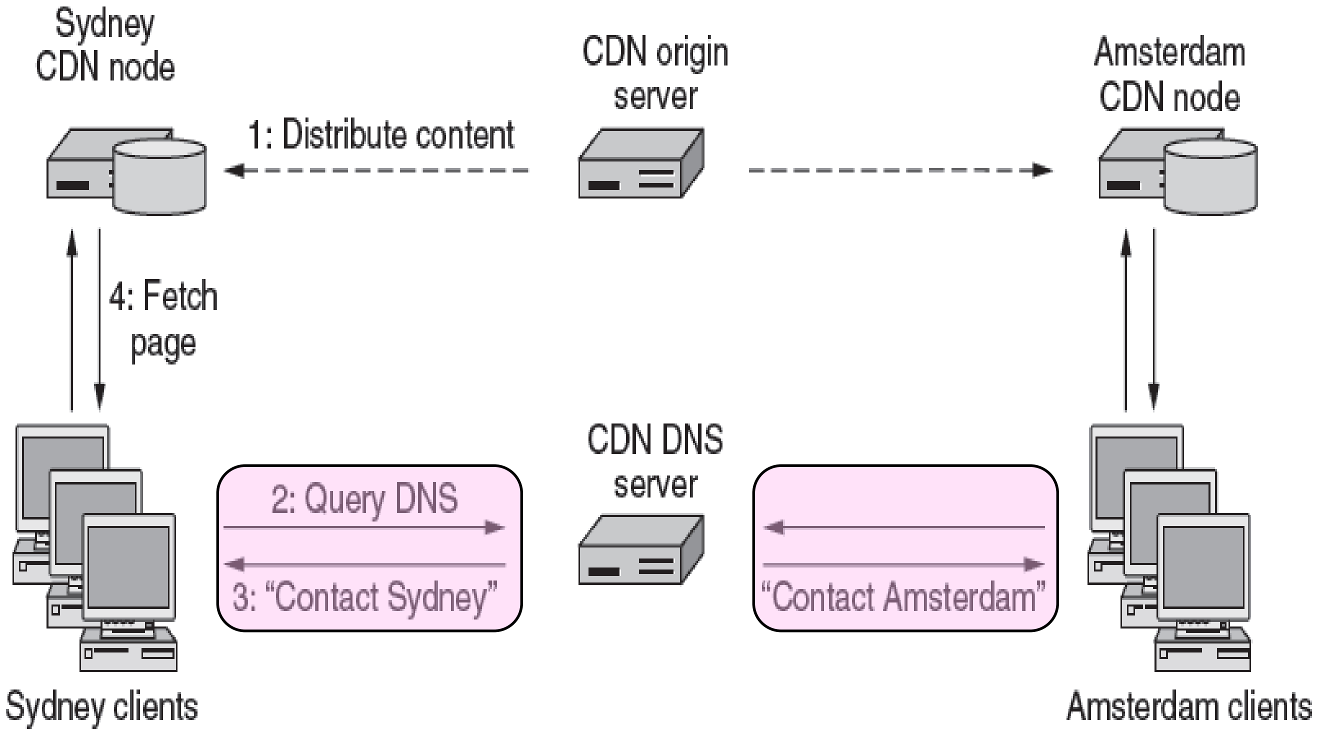
- Use browser and proxy caches
  - Helps, but limited to one client or clients in one organization
- Want to place replicas across the Internet for use by all nearby clients
  - Done by clever use of DNS

Content Delivery Network



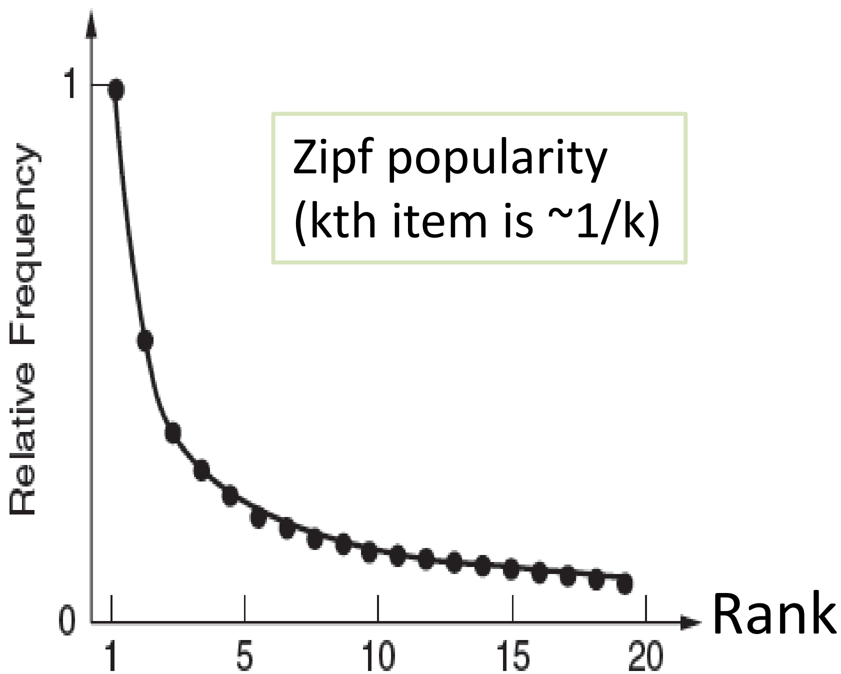
Content Delivery Network (2)

- DNS gives different answers to clients
  - Tell each client the nearest replica (map client IP)



Limits: Popularity of Content

- Zipf's Law: few popular items, many unpopular ones; both matter



George Zipf (1902-1950)

Source: Wikipedia