Overview of `HTTP` and Related Technologies

Lecture 04

Learning Outcomes

After completing this lesson, you should be able to

- Explore **different versions of HTTP protocol**
- Explain the **basic features of each of HTTP protocol versions**
- Explain the **limitations of each of the HTTP protocol versions**
- Explain the **HTTP Request/Response message components**
- Explain the **difference between safe and idempotent HTTP Request Methods**

HTTP

`HTTP` (**HyperText Transfer Protocol**) is the underlying protocol of the World Wide Web

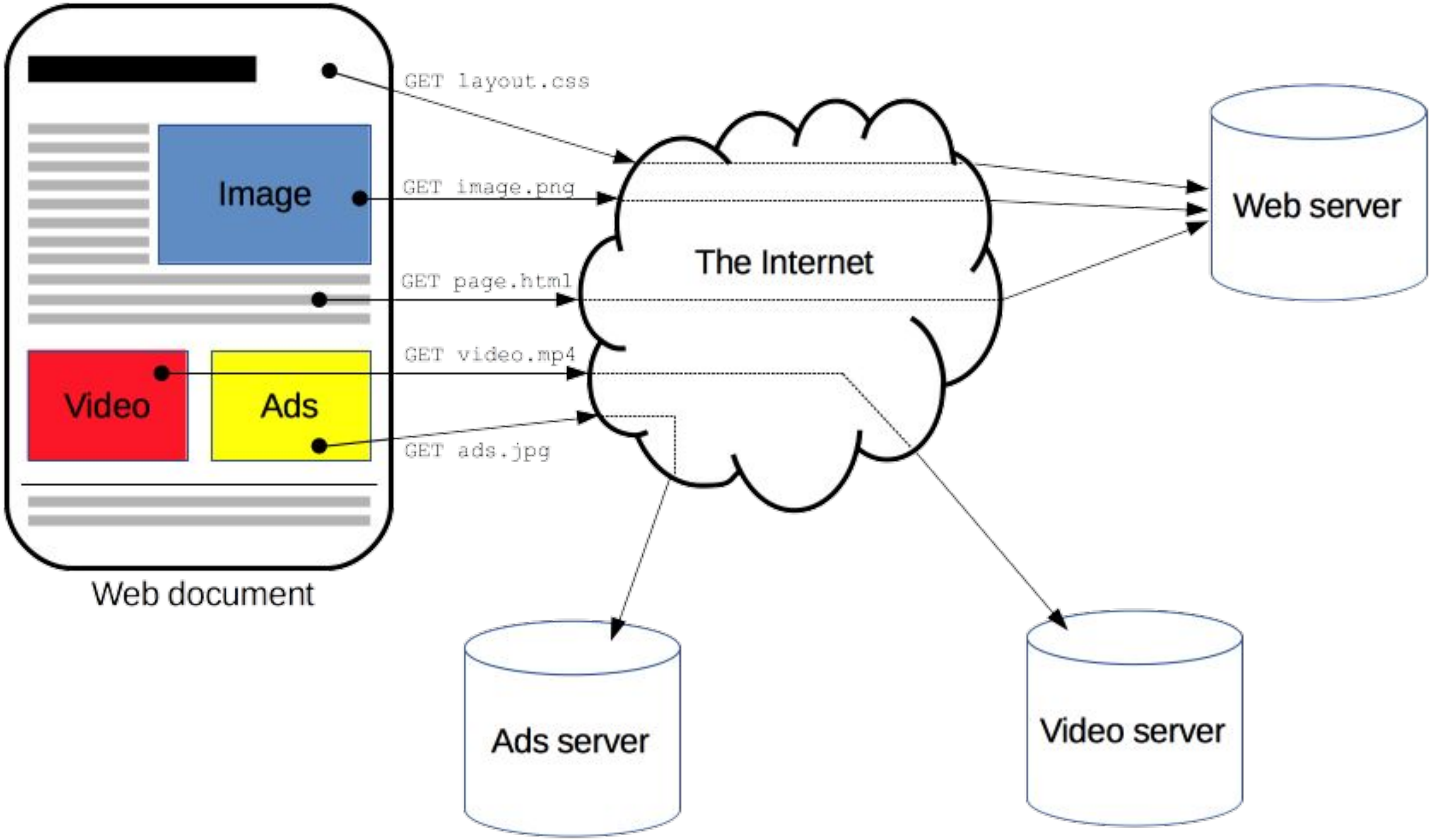
Built over TCP and IP protocols

It allows the fetching of resources, such as HTML documents

It is the foundation of any data exchange on the Web

it is a client-server protocol, which means requests are initiated by the recipient, usually the Web browser

HTTP

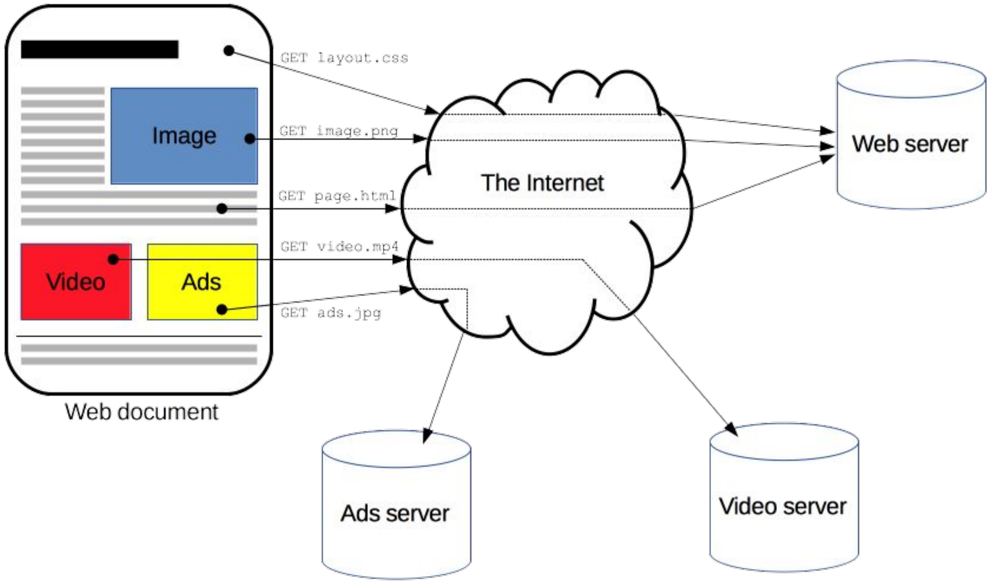


GET layout.css
GET image.png
GET page.html
GET video.mp4
GET ads.jpg
Image
Video
Ads
Web document
The Internet
Web server
Ads server
Video server

Evolution of HTTP

Below are list of different version of HTTP

- HTTP/0.9
- HTTP/1.0
- HTTP/1.1
- HTTP/2
- HTTP/3



`HTTP/0.9:` The one-line protocol

File transfer functionality,

Ability to request an index search of a hypertext archive,

Format negotiation, and

An ability to refer the client to another server

HTTP/0.9

Implemented Features

Client request is a single **ASCII** character string

Client request is terminated by a carriage return (**CRLF**)

Server response is an **ASCII** character stream

Server response is a hypertext markup language (**HTML**)

Connection is terminated after the document transfer is complete

HTTP/0.9

Example

The **request** consists of a **single line**

GET method and the path of the requested document.

The **response** is a **single hypertext document**

No headers or any other metadata, just the HTML

```
$> telnet google.com 80

Connected to 74.125.xxx.xxx

GET /about/

(hypertext response)
(connection closed)
```

HTTP/0.9

Client-server, request-response protocol

ASCII protocol, running over a TCP/IP link

Designed to transfer hypertext documents (HTML)

The connection between server and client is closed after every request.

Question

From what you already know about HTTP, **can you identify some of the limitations of the `HTTP/0.9` protocol ?**

`HTTP/0.9`: Limitations

Could not serve more documents than hypertext documents

It has **`GET`** request method only

Unable to provide metadata about the request and the response

Unable to negotiate content

`HTTP/1.0`: Building extensibility

HTTP Working Group (HTTP-WG) published **RFC 1945**, which **documented** the **"common usage"** of the many **`HTTP/1.0`** implementations found in the wild

`HTTP/1.0`: Features

Versioning information is now sent within each request (`HTTP/1.0` is appended to the `GET` line)

The notion of **HTTP headers** has been introduced, **both for the requests and the responses**, allowing metadata to be transmitted and making the protocol extremely flexible and extensible

`HTTP/1.0`: Features

Request and response **headers** were **`ASCII` encoded**

With the help of the new **`HTTP` headers**, the **ability to transmit other documents than plain HTML files** has been added (using the **`Content-Type`** header)

In addition to media type negotiation it included capabilities such as content encoding, character set support, multi-part types, authorization, caching, proxy behaviors, date formats, and more

The connection between server and client is closed after every request

HTTP/1.0 Example

1. **Request line** with HTTP version number, followed by **request headers**
2. **Response status**, followed by **response headers**

```
$> telnet website.org 80

Connected to xxx.xxx.xxx.xxx

GET /rfc/rfc1945.txt HTTP/1.0 (1)
User-Agent: CERN-LineMode/2.15 libwww/2.17b3
Accept: */*

HTTP/1.0 200 OK (2)
Content-Type: text/plain
Content-Length: 137582
Expires: Thu, 01 Dec 1997 16:00:00 GMT
Last-Modified: Wed, 1 May 1996 12:45:26 GMT
Server: Apache 0.84

(plain-text response)
(connection closed)
```

Question

Identify the

- Request Line
- Request Header
- Response Line
- Response Header

```
GET /mypage.html HTTP/1.0
User-Agent: NCSA_Mosaic/2.0 (Windows 3.1)

200 OK
Date: Tue, 15 Nov 1994 08:12:31 GMT
Server: CERN/3.0 libwww/2.17
Content-Type: text/html
<HTML>
A page with an image
  <IMG SRC="/myimage.gif">
</HTML>
```

Question

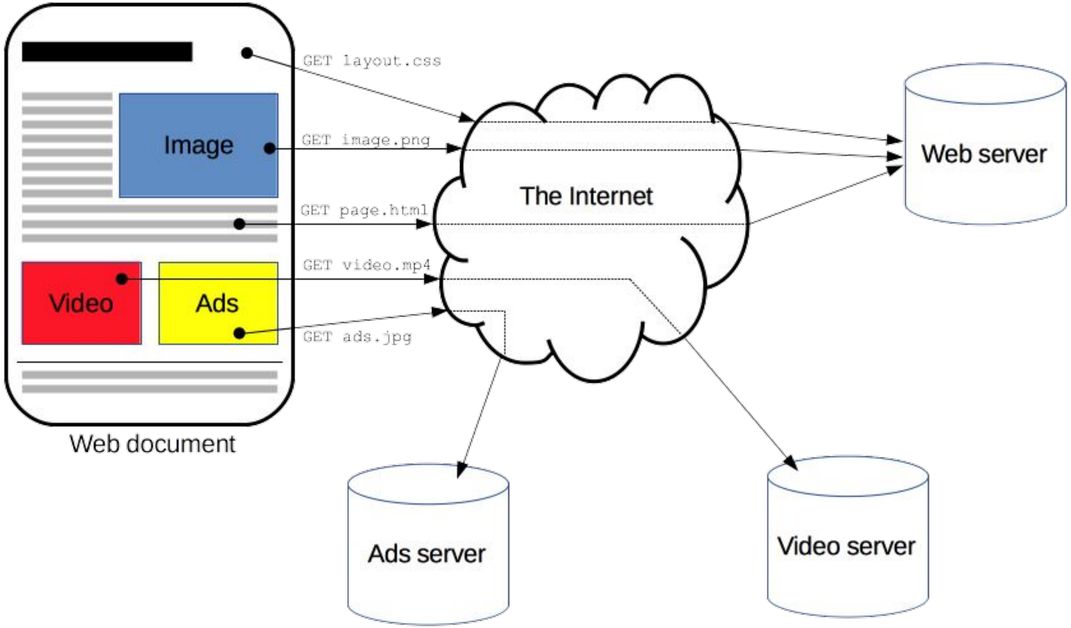
What kind of document is the client requesting in the following request?

```
GET /myimage.gif HTTP/1.0
User-Agent: NCSA_Mosaic/2.0 (Windows 3.1)

200 OK
Date: Tue, 15 Nov 1994 08:12:32 GMT
Server: CERN/3.0 libwww/2.17
Content-Type: text/gif
(image content)
```

Question

What do you think are **the limitations of the `HTTP/1.0` protocol ?**



`HTTP/1.0`: Limitations

Requiring a new TCP connection per request imposes a significant performance penalty

`HTTP/1.1`: Standardized protocol

The official **`HTTP/1.1`** standard is defined in **`RFC 2068`**, which was released in January 1997

In June of 1999, a number of improvements and updates were incorporated into the standard and the second version were released as **`RFC 2616`**

The **`HTTP/1.1`** standard resolved a lot of the protocol ambiguities found in earlier versions

HTTP/1.1

It introduced a number of critical performance optimizations:

Keepalive Connections

A connection can be reused, saving the time to reopen it numerous times to display the resources embedded into the single original document retrieved

Chunked Encoding Transfers

Chunked responses are now also supported

HTTP/1.1

It introduced a number of critical performance optimizations:

Byte-range Requests

Request Pipelining has been added, allowing to send a second request before the answer for the first one is fully transmitted, lowering the latency of the communication

HTTP/1.1

It introduced a number of critical performance optimizations:

Additional cache control mechanisms have been introduced

Content negotiation, including language, encoding, or type, has been introduced

The addition of the **Host header** allowed to host different domains at the same IP address (allowing server collocation)

HTTP/1.1: Example

1 Request for HTML file, with encoding metadata

```
$> telnet website.org 80
Connected to xxx.xxx.xxx.xxx

GET /index.html HTTP/1.1 1
Host: website.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4)... (snip)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
```

HTTP/1.1: Example

2 Chunked response for original HTML request

```
HTTP/1.1 200 OK 2
Server: nginx/1.0.11
Connection: keep-alive
Content-Type: text/html; charset=utf-8
Via: HTTP/1.1 GWA
Date: Wed, 25 Jul 2012 20:23:35 GMT
Expires: Wed, 25 Jul 2012 20:23:35 GMT
Cache-Control: max-age=0, no-cache
Transfer-Encoding: chunked
```

HTTP/1.1: Example

3 Number of octets in the chunk expressed as an ASCII hexadecimal number

4 End of chunked stream response

```
100 3
<!doctype html>
(snip)

100
(snip)

0 4
```

HTTP/1.1: Example

5 Request for an icon file made on same TCP connection

6 Inform server that the connection will not be reused

```
GET /favicon.ico HTTP/1.1 5
Host: www.website.org
User-Agent: Mozilla/5.0 (Macintosh; I
Accept: */*
Referer: http://website.org/
Connection: close 6
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.
Cookie: __qca=P0-800083390... (snip)
```

HTTP/1.1: Example

7 Icon response, followed by connection close

```
HTTP/1.1 200 OK 7
Server: nginx/1.0.11
Content-Type: image/x-ico
Content-Length: 3638
Connection: close
Last-Modified: Thu, 19 Ju
Cache-Control: max-age=31
Accept-Ranges: bytes
Via: HTTP/1.1 GWA
Date: Sat, 21 Jul 2012 21
Expires: Thu, 31 Dec 2037
Etag: W/PSA-GAu26oXbDi

(icon data)
(connection closed)
```

Question

What is the name of the header part that is added on **`HTTP/1.1`** version to improve the limitation of **`HTTP/1.0`** (requiring a new TCP connection for each request)

`HTTP/1.1`: Example Summary

We have two object requests, one for an HTML page and one for an image, both delivered over a single connection.

This is connection keepalive in action, which allows us to reuse the existing TCP connection for multiple requests to the same host and deliver a much faster end-user experience

`HTTP/1.1`: Example Summary

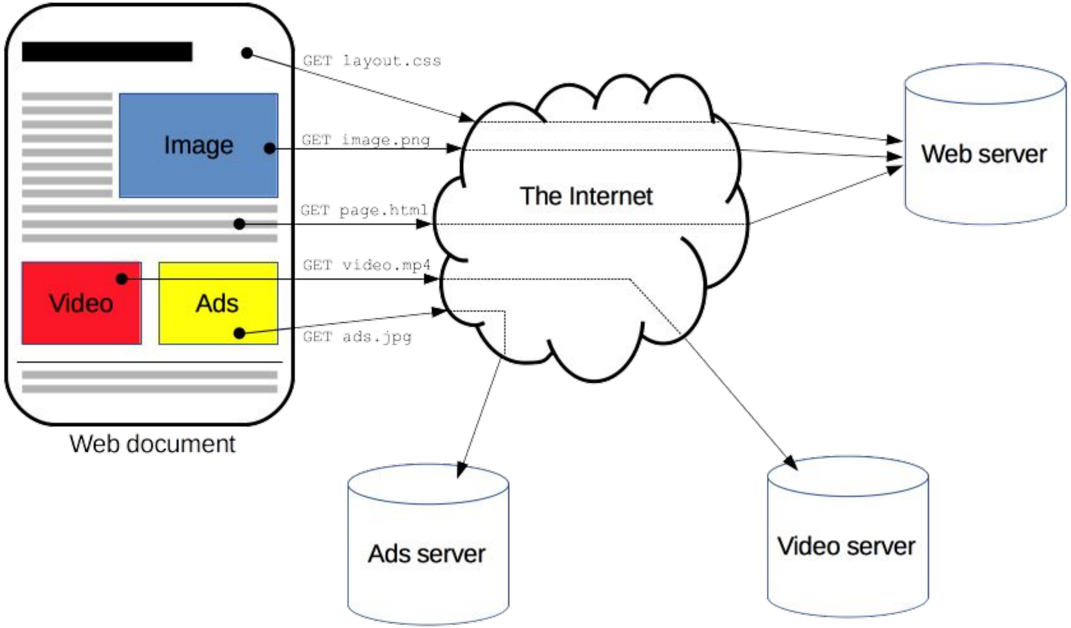
To terminate the persistent connection, notice that the second client request sends an explicit close token to the server via the Connection header.

Similarly, the server can notify the client of the intent to close the current TCP connection once the response is transferred

`HTTP/1.1` changed the semantics of the HTTP protocol to use connection keepalive by default. Meaning, unless told otherwise (via Connection: close header), the server should keep the connection open by default

Question

What do you think are **the limitations of the `HTTP/1.X` protocol ?**



Limitations of `HTTP/1.x`

Clients need to use **multiple connections to achieve concurrency** and **reduce latency**

Does not compress request and response **headers**, causing unnecessary network traffic

Does not allow effective **resource prioritization**, resulting in poor use of the underlying TCP connection

SPDY

SPDY was an experimental protocol, developed at Google and announced in mid-2009

Its primary goal was to try to **reduce the load latency of web pages** by addressing some of the well-known performance limitations of `HTTP/1.1`

SPDY

The **specific** project **goals** were the following

- Target a **50% reduction in page load time (PLT)**
- Avoid the need for any changes to content by website authors
- Minimize deployment complexity, avoid changes in network infrastructure
- Develop this new protocol in partnership with the open-source community
- Gather real performance data to (in)validate the experimental protocol

SPDY

SPDY in lab condition has shown **55% reduction in page load time**

As a result SPDY was supported in Chrome, Firefox, and Opera, and a rapidly growing number of sites, both large (e.g., Google, Twitter, Facebook) and small

In effect, SPDY was on track to become a de facto standard through growing industry adoption

SPDY and HTTP/2

Observing the trend, the **`HTTP Working Group (HTTP-WG)`** kicked off a new effort

to take the lessons learned from SPDY,

to build and improve on them, and

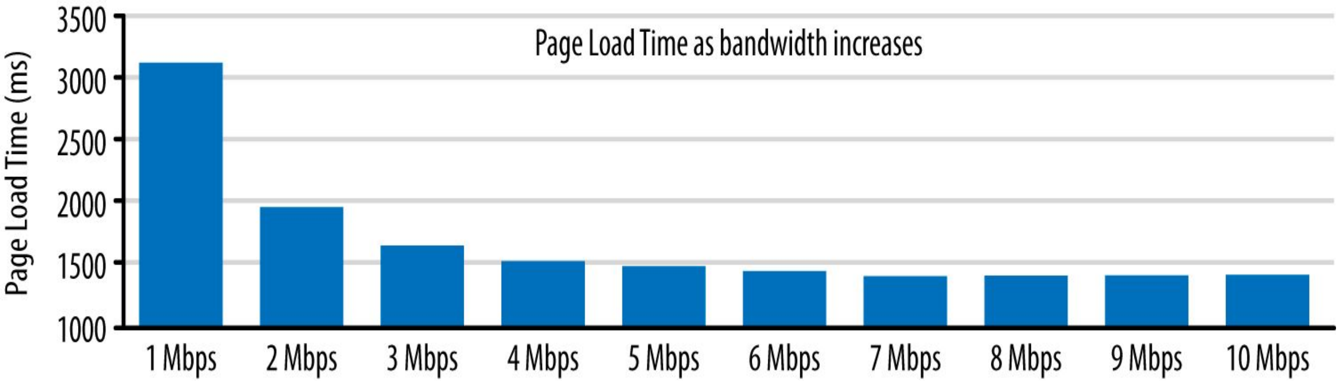
to deliver an official "**`HTTP/2`**" standard

HTTP/2

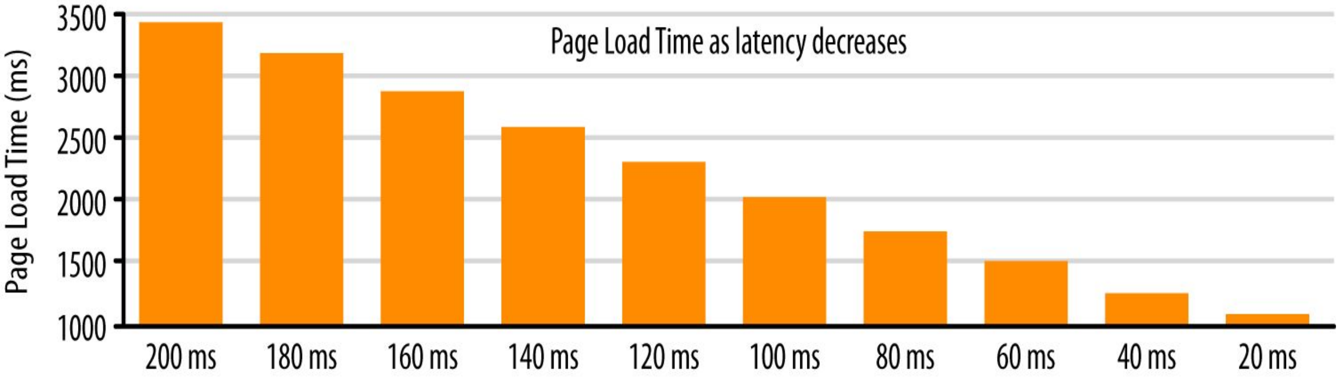
HTTP/2 is a protocol designed for **low-latency transport of content** over the World Wide Web

- **Improve end-user perceived latency**
- **Address the "head of line blocking"**
- **Not require multiple connections**
- **Retain the semantics of HTTP/1.1**

Latency vs Bandwidth impact on Page Load Time



Single digit % perf improvement after 5 Mbps



Linear improvement in page load time!

Latency vs Bandwidth impact on Page Load Time

Decreasing latency has more impact than increasing bandwidth

For Example

> Decreasing RTTs from 150 ms to 100 ms have a larger effect on the speed of the internet than increasing a user's bandwidth from 3.9 Mbps to 10 Mbps or even 1 Gbps

HTTP/2 : Streams, Messages, and Frames

The introduction of the new binary framing mechanism changes how the data is exchanged between the client and server

Stream

A bidirectional flow of bytes within an established connection, which may carry one or more messages

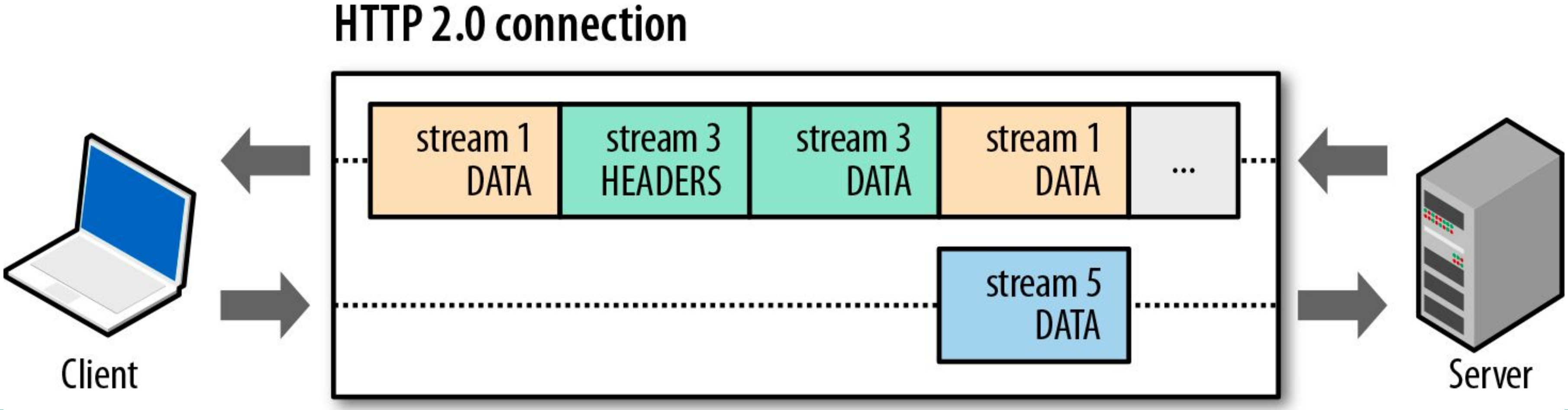
Message

A complete sequence of frames that map to a logical request or response message

HTTP/2: Streams, Messages, and Frames

Frame

The smallest unit of communication in **HTTP/2**, each containing a frame header, which at a minimum identifies the stream to which the frame belongs



`HTTP/2`: Streams, Messages, and Frames

The **frame is the smallest unit of communication** that carries a specific type of data—e.g., **HTTP headers**, **message payload**, and so on.

Frames from different streams may be **interleaved** and then **reassembled** via the embedded **stream identifier** in the header of each frame

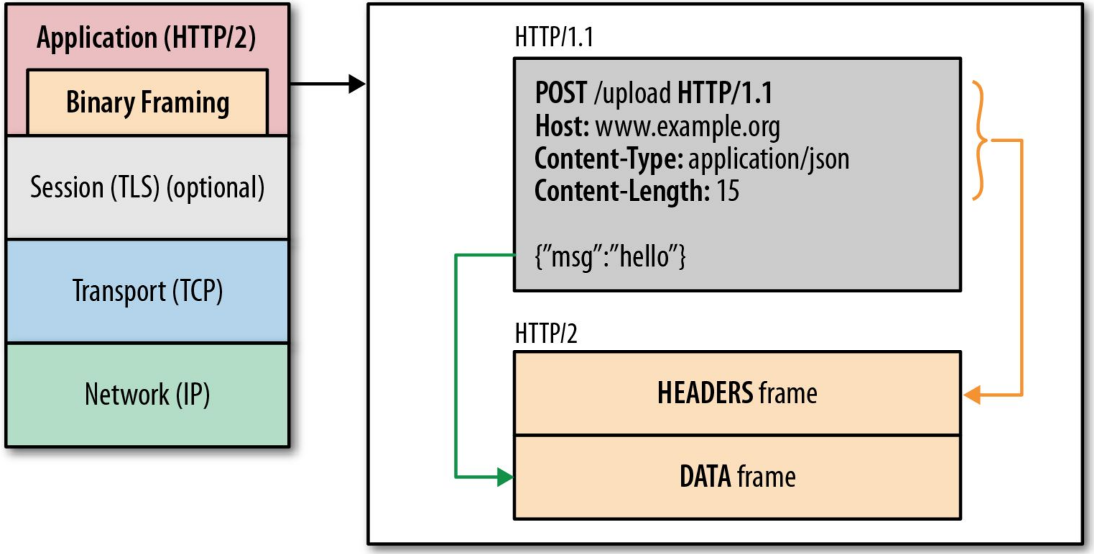
`HTTP/2` breaks down the HTTP protocol communication into an exchange of **binary-encoded frames**, which are then mapped to **messages** that belong to a particular **stream**, and all of which are **multiplexed** within a **single TCP connection**

HTTP/2: Main Characteristics

One TCP connection

Request → Stream

- **Streams are multiplexed**
- **Streams are prioritized**

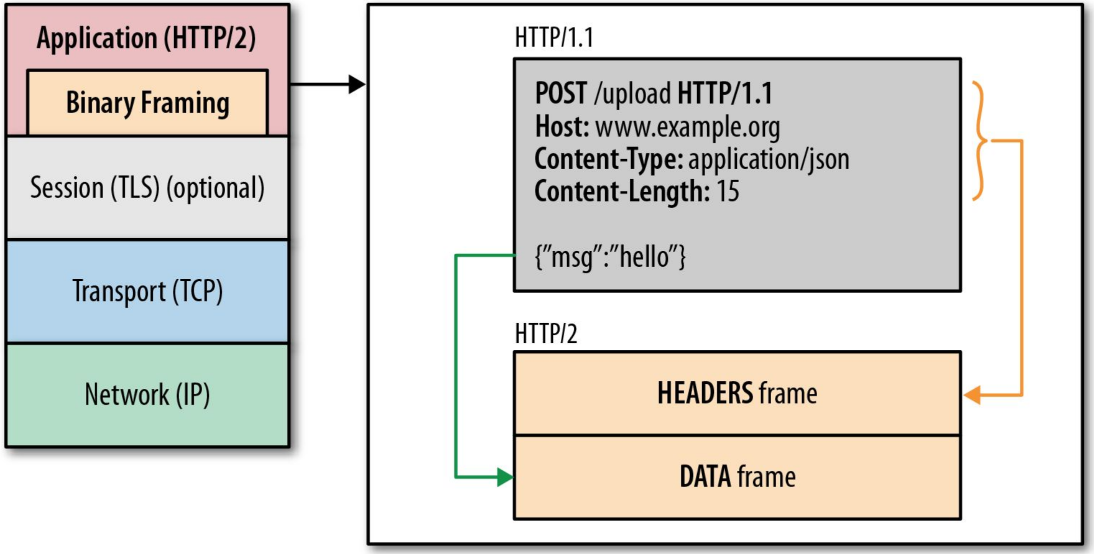


HTTP/2: Main Characteristics

Binary framing layer

- Prioritization
- Flow control
- Server push

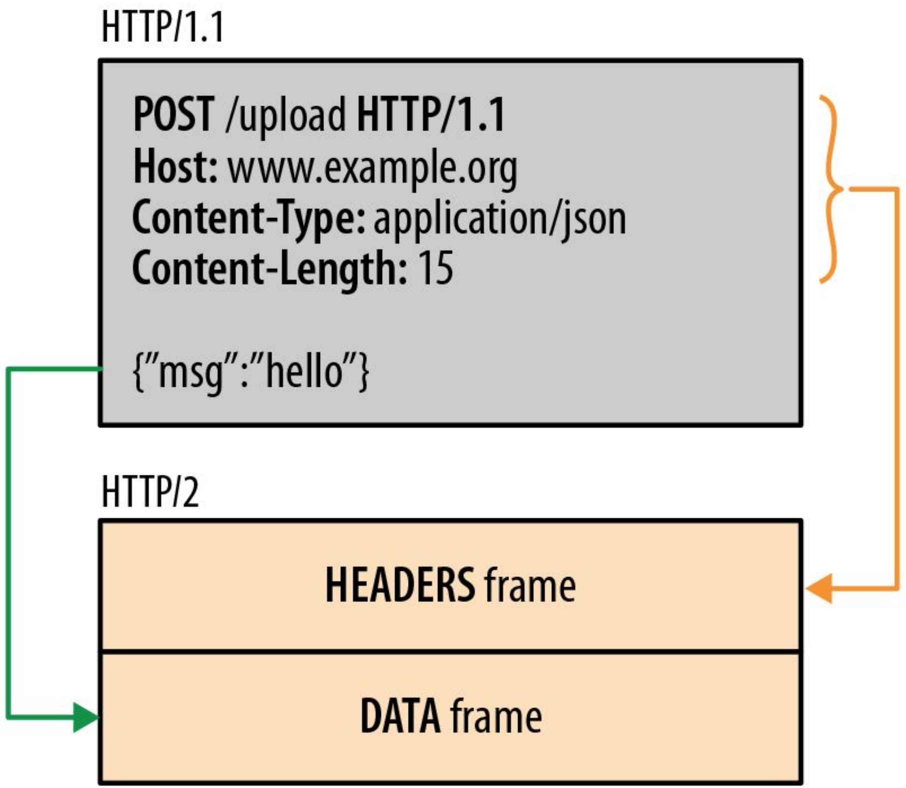
Header compression (HPACK)



HTTP/2: Binary framing

HTTP messages are decomposed into one or more frames

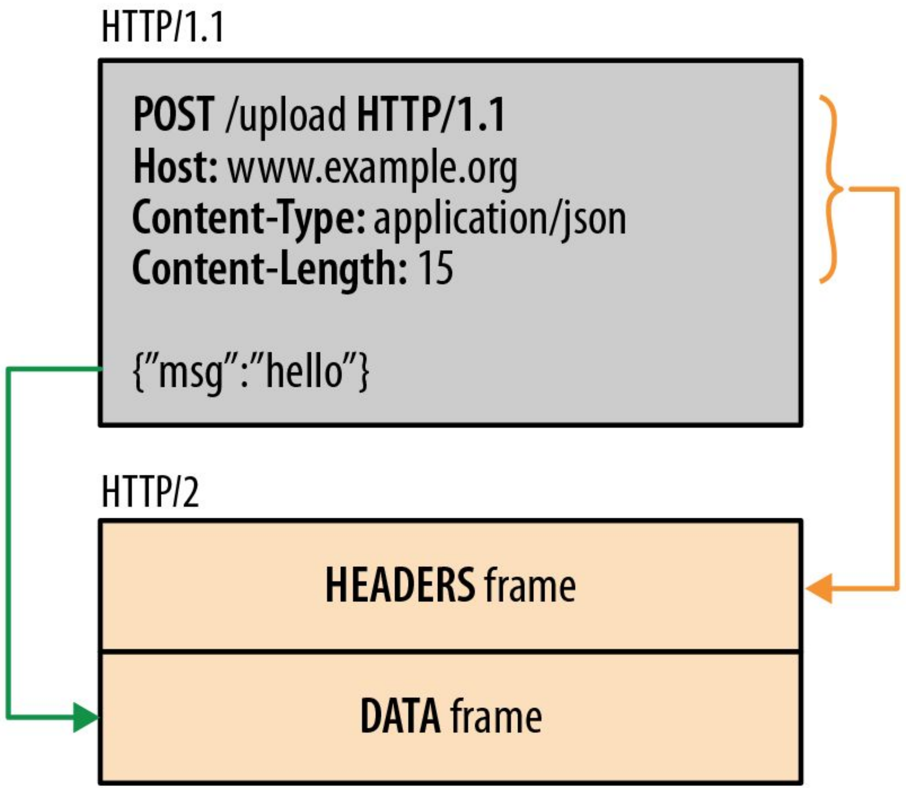
- **HEADERS** for meta-data
- **DATA** for payload
- **RST_STREAM** to cancel



HTTP/2: Binary framing

Each frame has a common header

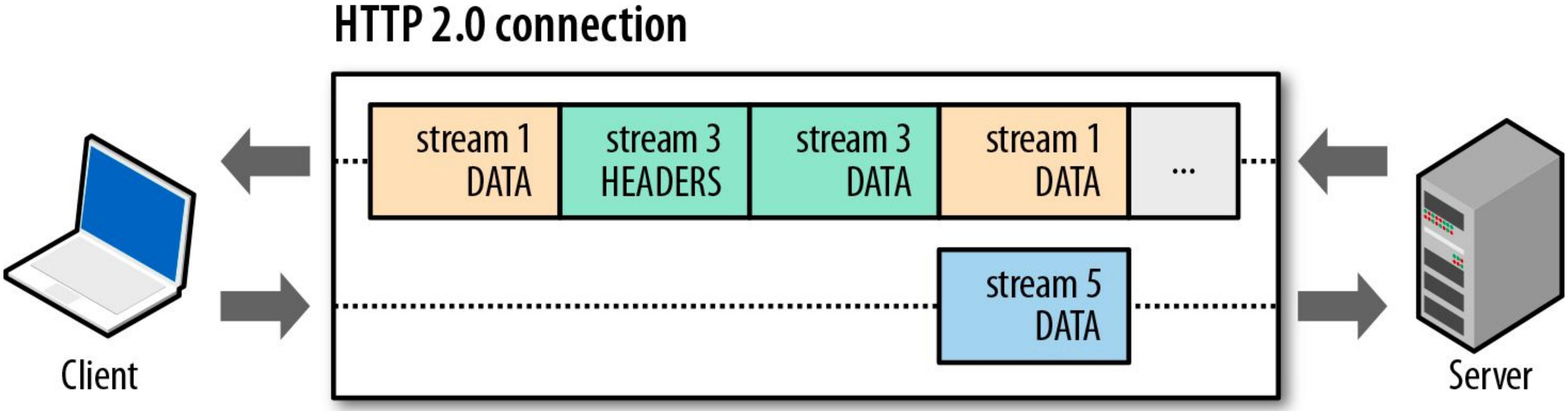
- 9-byte, length prefixed
- Easy and efficient to parse



HTTP/2: Basic data flow

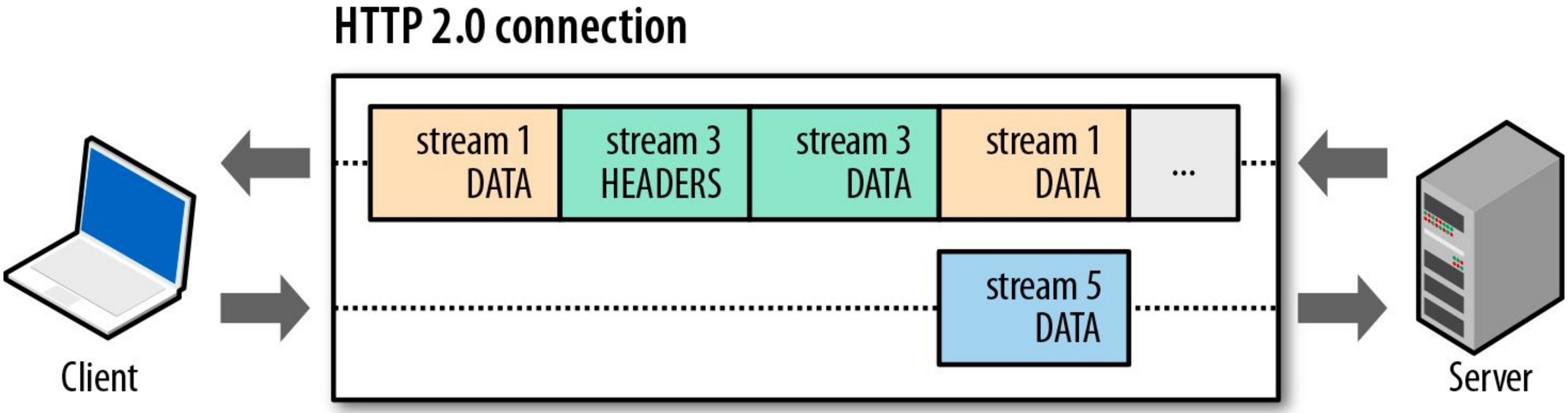
How many streams are there in the diagram?

How many frames?



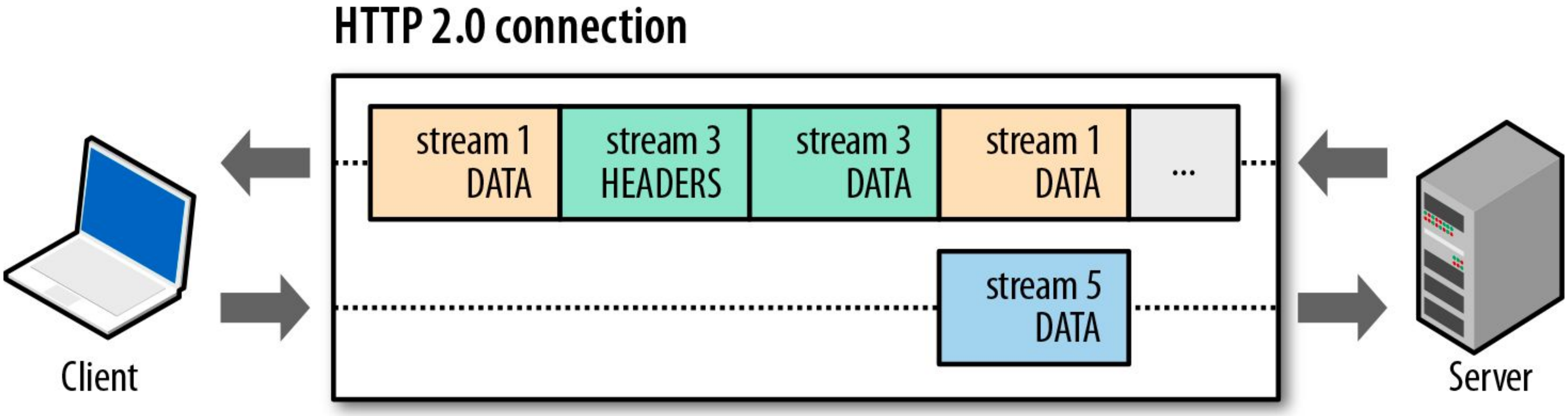
`HTTP/2:` Stream Multiplexing

Streams are multiplexed because frames can be interleaved



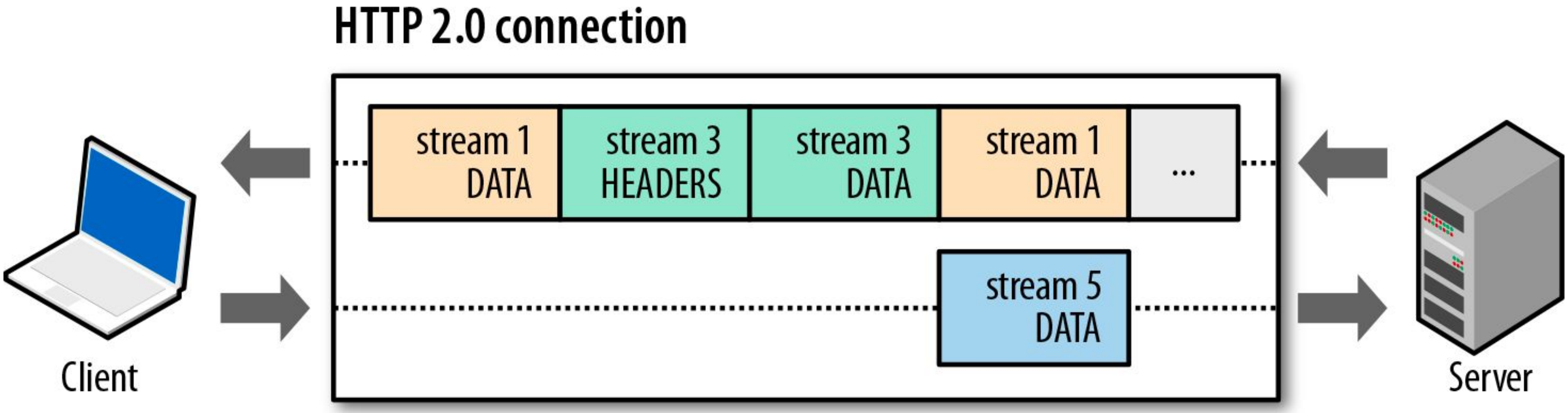
HTTP/2: Stream Priotization

Streams are priotized based on their **weight** and **dependency**



HTTP/2: Stream Multiplexing

What are the advantages of stream multiplexing?



`HTTP/2`: Stream Multiplexing

Advantages

- **Interleave multiple requests in parallel** without blocking on any one
- **Interleave multiple responses in parallel** without blocking on any one
- Use a **single connection to deliver multiple requests** and responses in parallel

`HTTP/2:` Stream Multiplexing

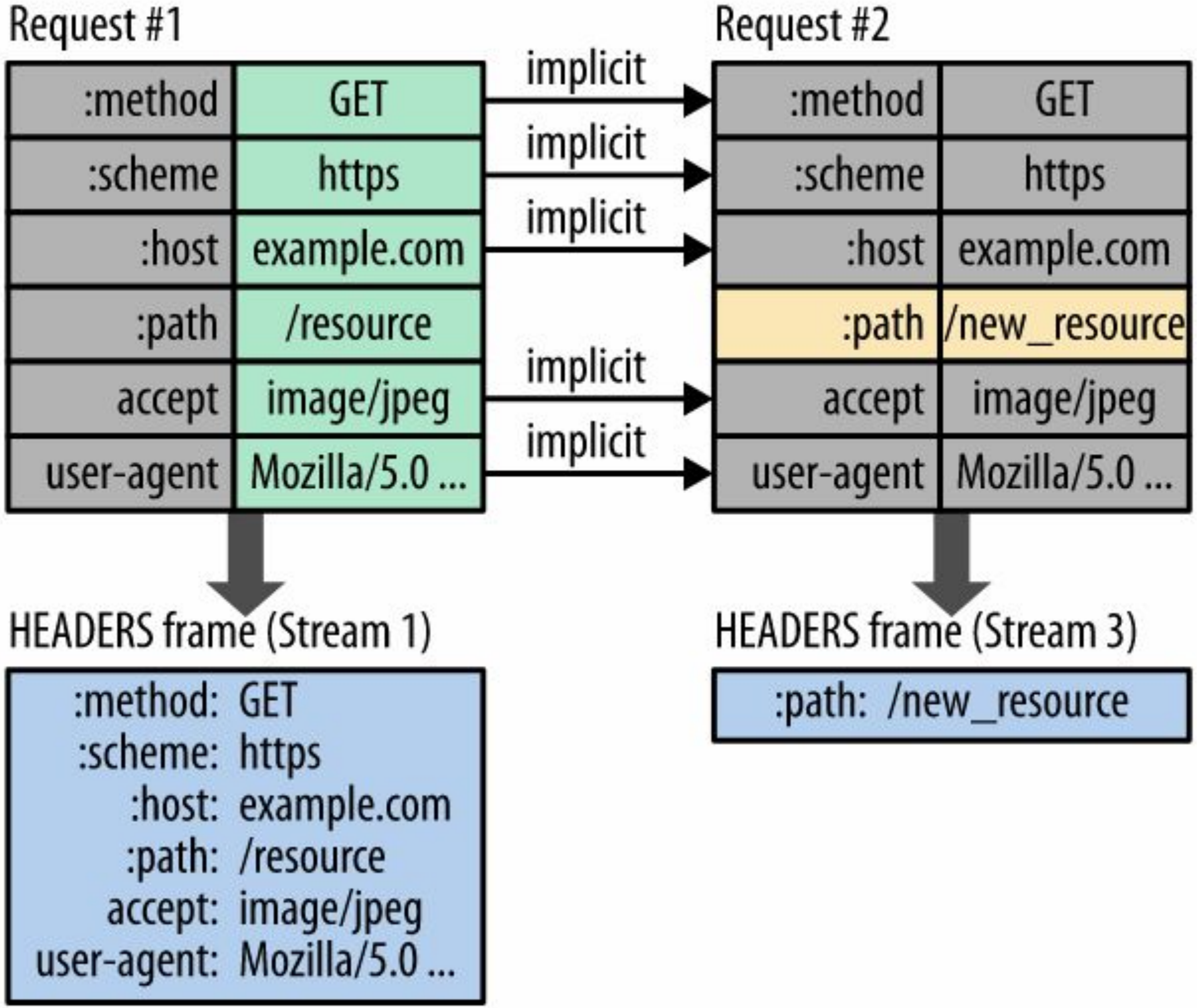
Advantages

- **Remove unnecessary HTTP/1.x workarounds for optimization**, such as concatenated files, image sprites, and domain sharding
- Deliver **lower page load times** by eliminating unnecessary latency and improving utilization of available network capacity

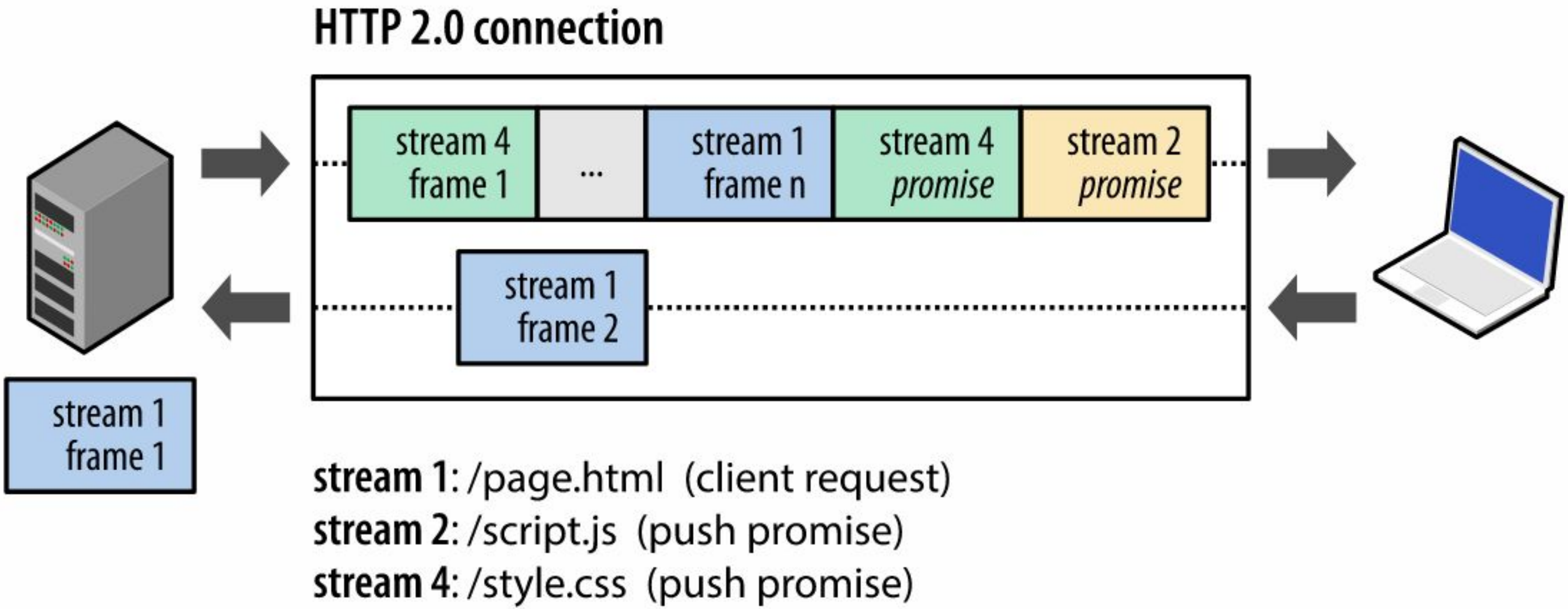
HTTP/2

Header Compression

Uses `HPACK` algorithm

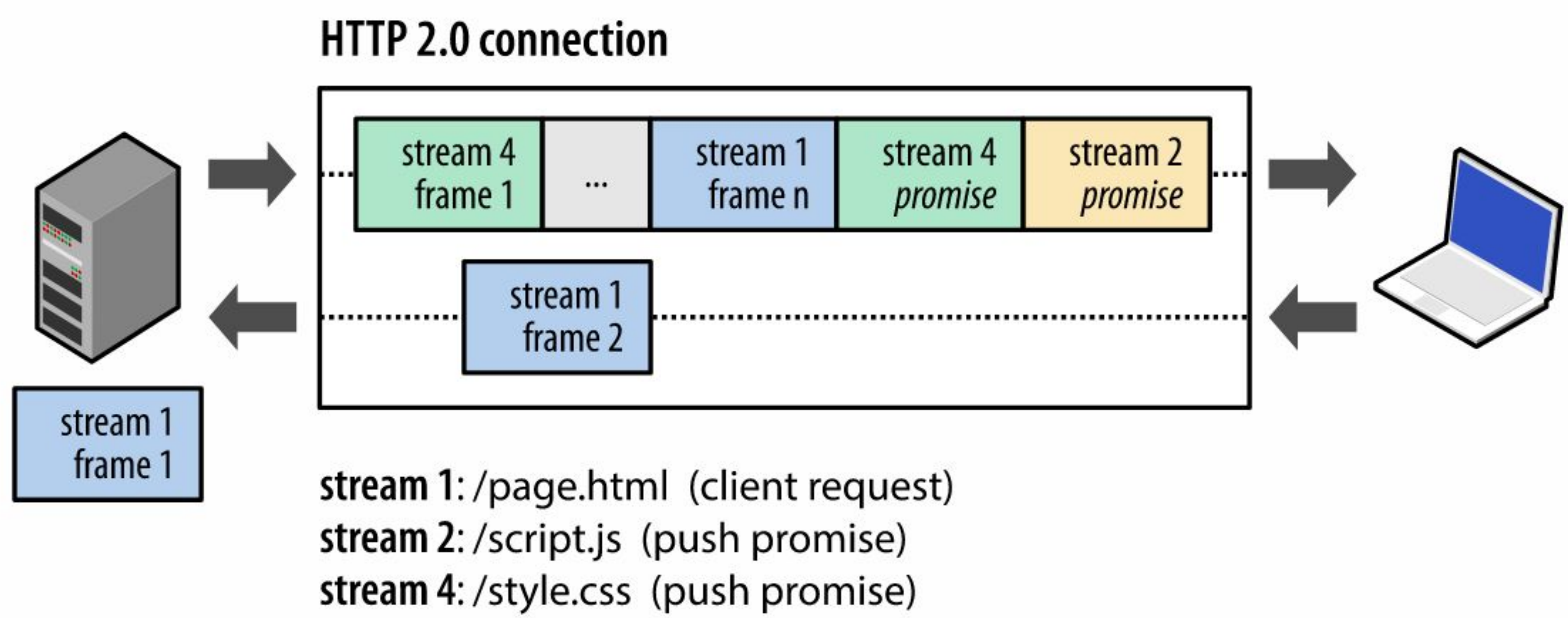


HTTP/2: Server Push



HTTP/2: Server Push

What are the advantages of server push?



HTTP/2: Server Push

Benefits

- Pushed resources can be cached by the client
- Pushed resources can be reused across different pages
- Pushed resources can be multiplexed alongside other resources
- Pushed resources can be prioritized by the server

HTTP/2 : Flow Control

A mechanism to **prevent the sender from overwhelming the receiver with data** it may not want or be able to process as

the receiver may be busy,

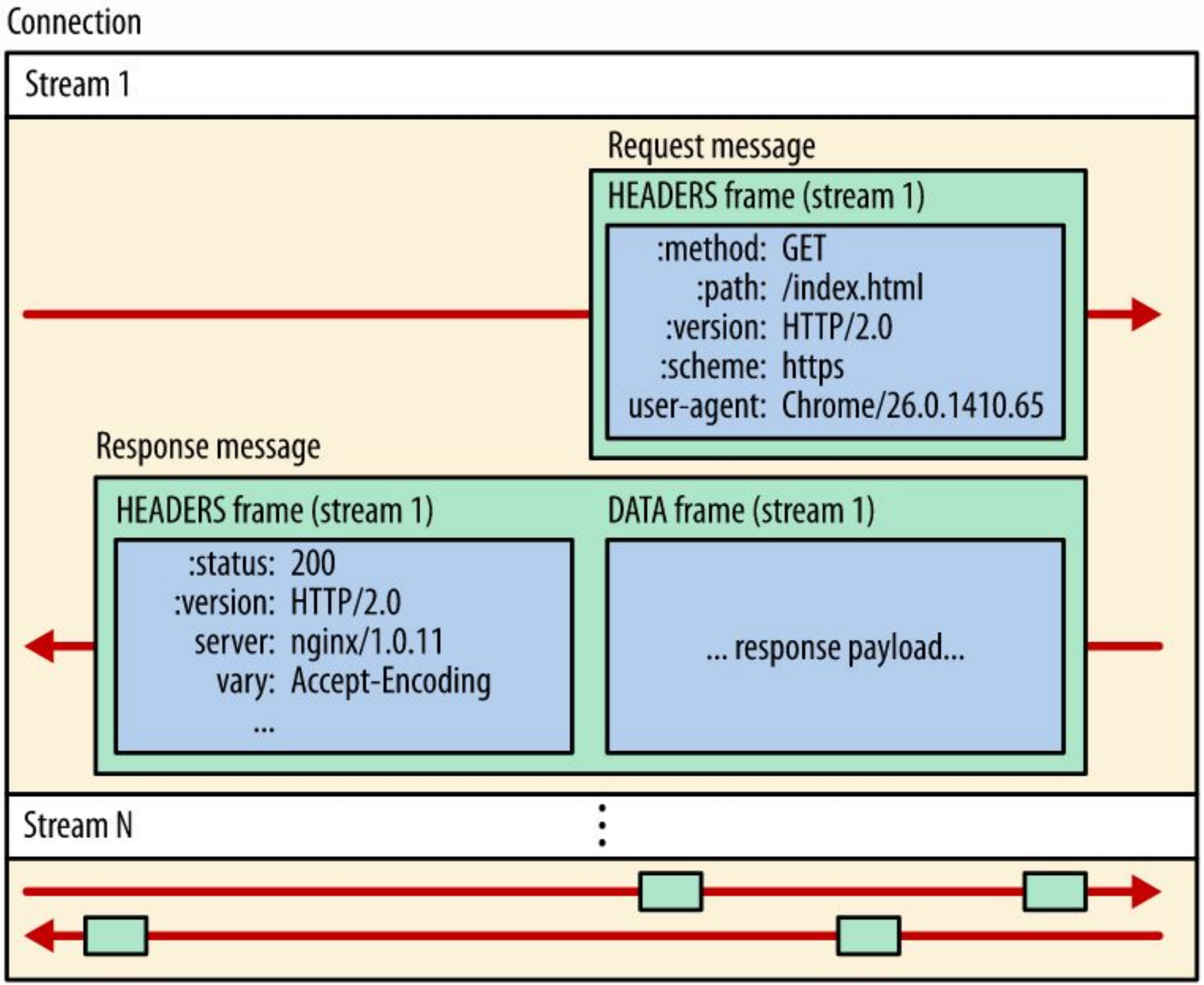
under heavy load, or

may only be willing to allocate a fixed amount of resources for a particular stream

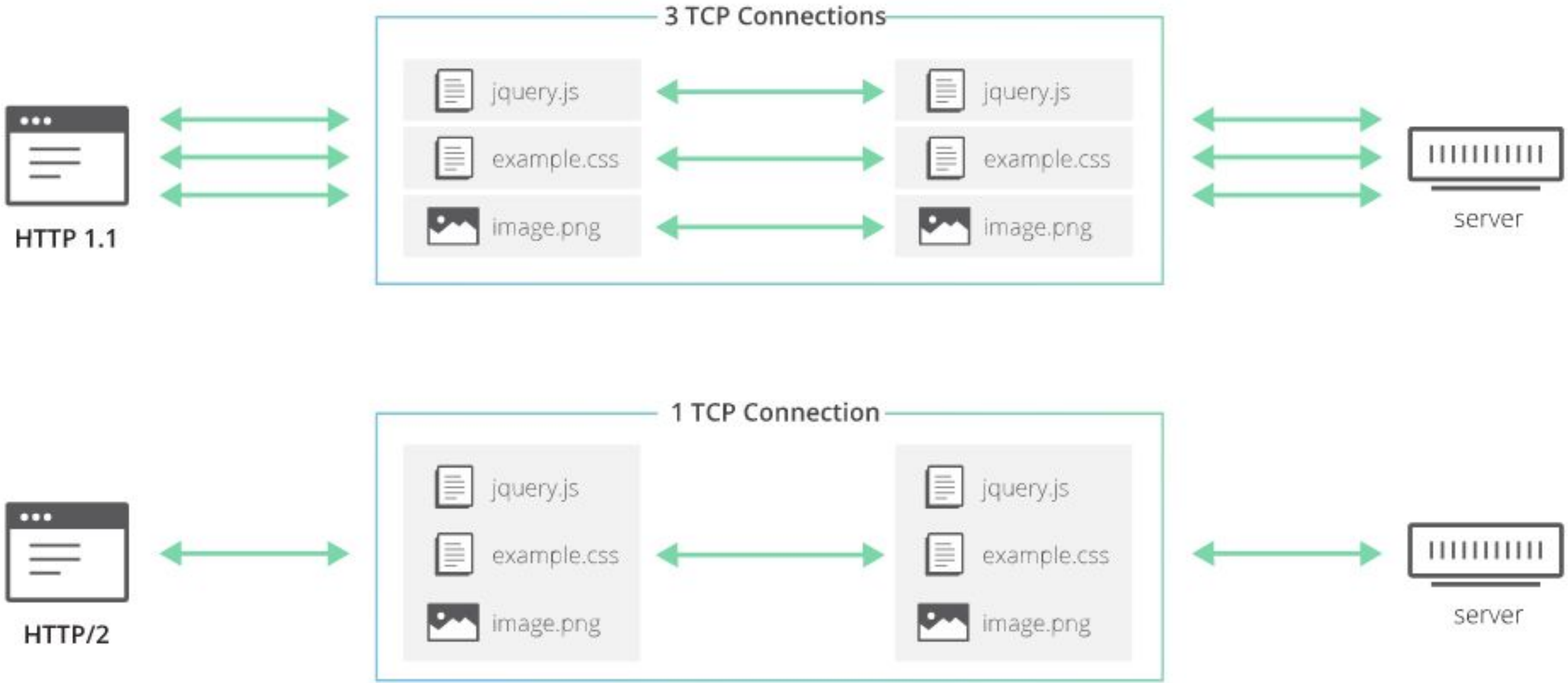
HTTP/2

Single TCP

Multiple Stream

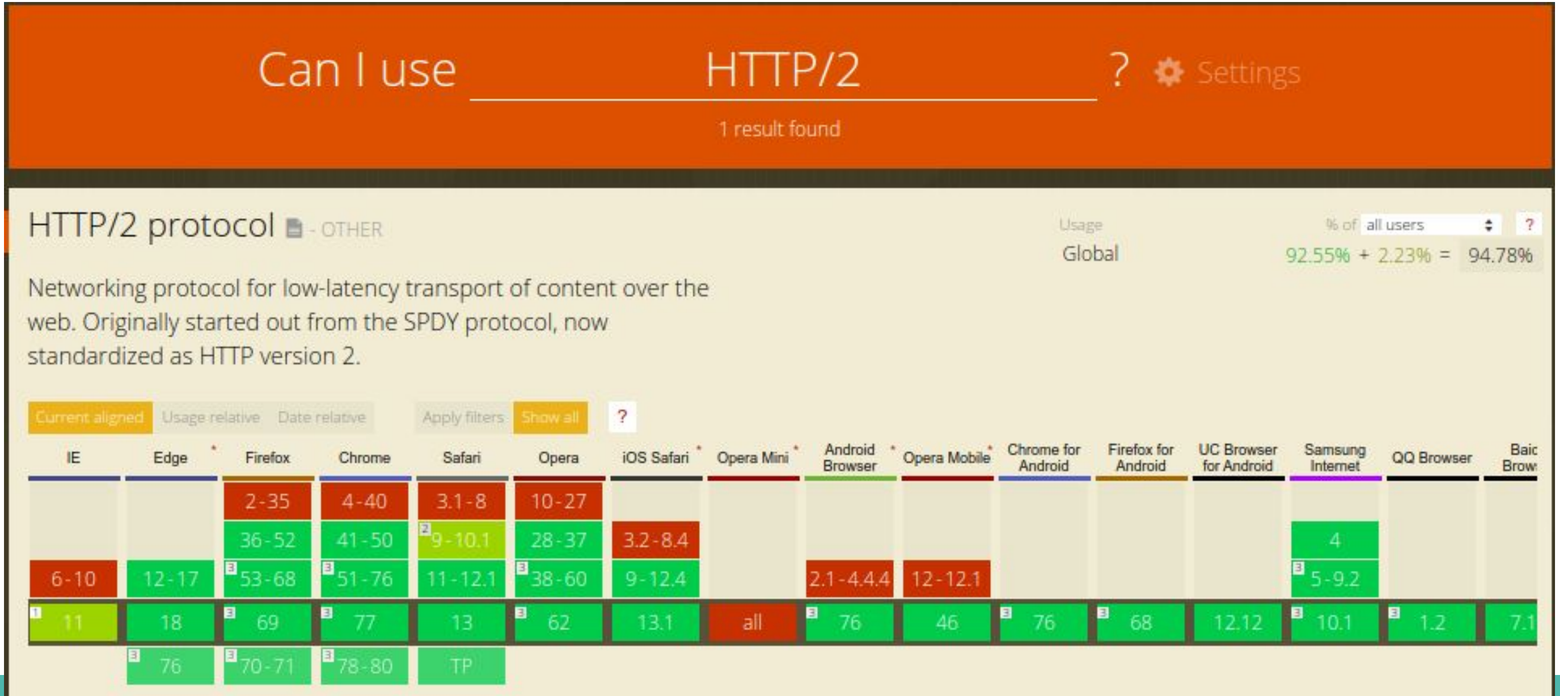


HTTP/1.X vs HTTP/2 TCP Connection



Browsers Supporting HTTP/2

https://caniuse.com



Limitations of `HTTP/2`

Do you think there is limitation to **HTTP/2** protocol?

QUIC

`QUIC` (**Q**uick **U**DP **I**nternet **C**onnections) is a new transport protocol for the internet, developed by Google

QUIC solves a number of transport-layer and application-layer problems experienced by modern web applications, while requiring little or no change from application writers.

QUIC is very similar to **TCP+TLS+HTTP2**, but **implemented on top of UDP**

QUIC

Key **advantages** of QUIC over TCP+TLS+HTTP2 include:

Low connection establishment latency

Improved congestion control

Multiplexing without head-of-line blocking

Forward error correction

Connection migration

`QUIC:` Low Connection establishment latency

QUIC handshakes frequently require **zero roundtrips before sending payload**, as compared to 1-3 roundtrips for TCP+TLS

QUIC: Congestion Control

QUIC has pluggable congestion control, and provides richer information to the congestion control algorithm than TCP.

Currently, Google’s implementation of QUIC uses a reimplementation of TCP Cubic and is experimenting with alternative approaches

`QUIC`: Multiplexing

One of the larger issues with **`HTTP/2`** on top of TCP is the **issue of head-of-line blocking**

When a TCP packet is lost, no streams on that **`HTTP2`** connection can make forward progress until the packet is retransmitted and received by the other side - not even when the packets with data for these streams have arrived and are waiting in a buffer

QUIC : Multiplexing

Lost packets carrying data for an individual stream generally only impact that specific stream

Each stream frame can be immediately dispatched to that stream on arrival, so streams without loss can continue to be reassembled and make forward progress in the application

QUIC: Forward Error Correction

QUIC can complement a group of packets with an FEC packet, In order to recover from lost packets without waiting for a retransmission.

Much like RAID-4, the FEC packet contains parity of the packets in the FEC group.

If one of the packets in the group is lost, the contents of that packet can be recovered from the FEC packet and the remaining packets in the group

QUIC: Connection Migration

QUIC connections are identified by a 64 bit connection ID, randomly generated by the client.

In contrast, **TCP connections are identified by** a **4-tuple** of **source address**, **source port**, **destination address** and **destination port**.

This means that if a client changes IP addresses (for example, by moving out of Wi-Fi range and switching over to cellular) or ports (if a NAT box loses and rebinds the port association), any active TCP connections are no longer valid. **When a QUIC client changes IP addresses, it can continue to use the old connection ID from the new IP address without interrupting any in-flight requests**

HTTP/3: HTTP over QUIC

Instead of using TCP as the transport layer for the session, it uses QUIC

QUIC introduces streams as first-class citizens at the transport layer.

QUIC streams share the same QUIC connection, so no additional handshakes and slow starts are required to create new ones

QUIC streams are delivered independently such that in most cases packet loss affecting one stream doesn't affect others.

This is possible because QUIC packets are encapsulated on top of UDP datagrams

HTTP/3: HTTP over QUIC

Using UDP allows much more flexibility compared to TCP, and enables QUIC implementations to live fully in user-space — updates to the protocol's implementations are not tied to operating systems updates as is the case with TCP

QUIC also combines the typical 3-way TCP handshake with **TLS 1.3**'s handshake

Combining these steps means that encryption and authentication are provided by default, and also enables faster connection establishment

`HTTP/3`: HTTP over QUIC

Check the browsers supporing `QUIC` or `HTTP/3`

HTTP Messages

HTTP messages, as defined in HTTP/1.1 and earlier, are human-readable

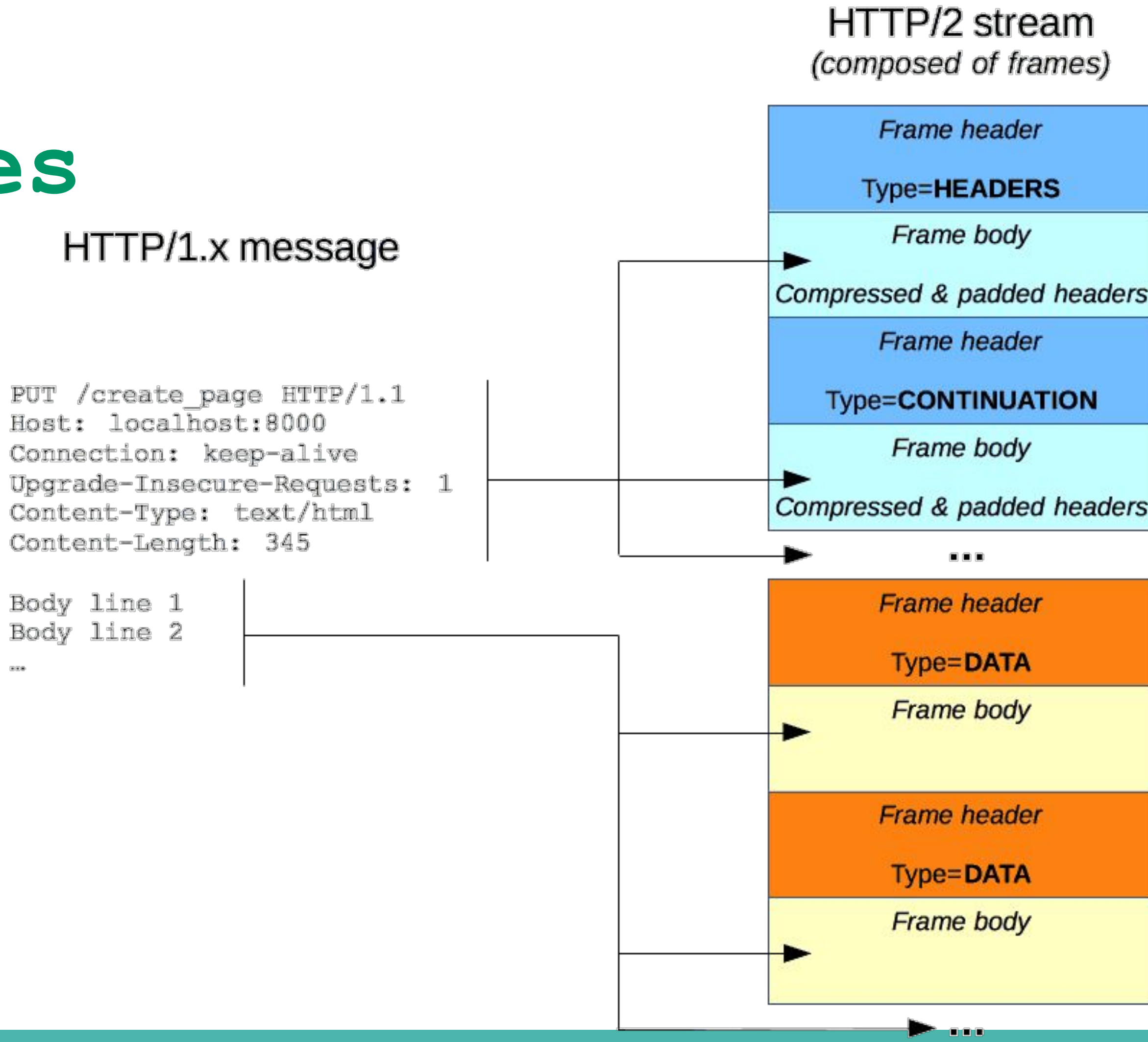
In HTTP/2, these messages are embedded into a binary structure, a frame, allowing optimizations like compression of headers and multiplexing.

Even if only part of the original HTTP message is sent in HTTP/2, **the semantics of each message is unchanged** and the client reconstitutes the original HTTP/1.1 request

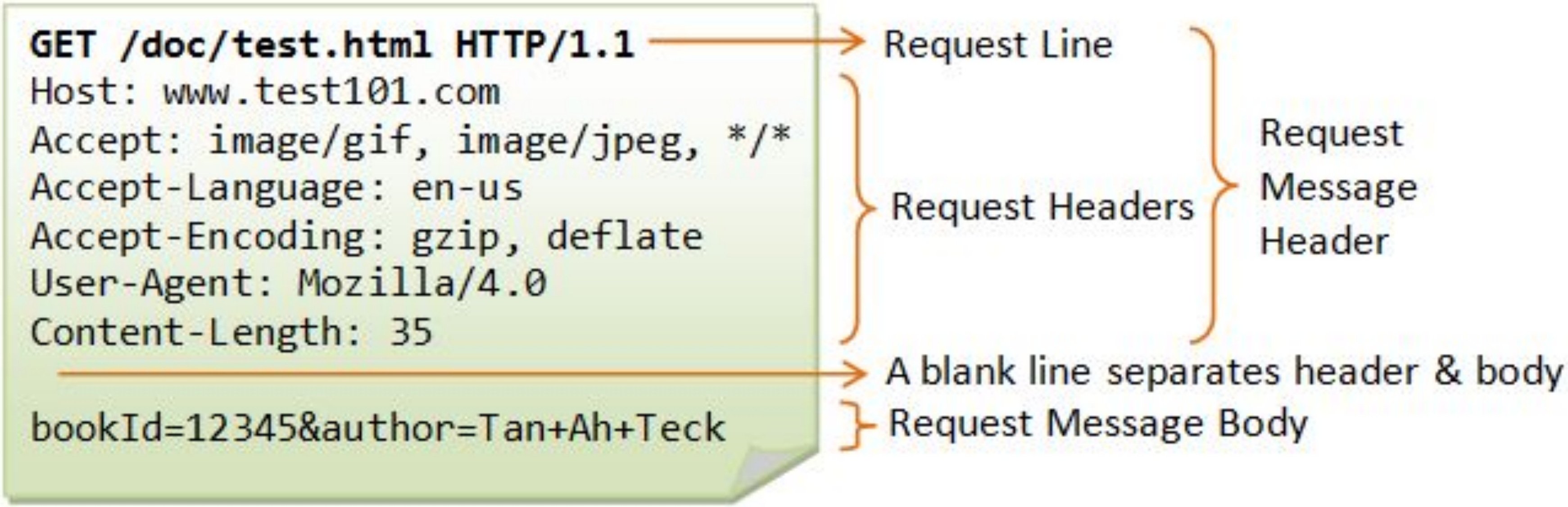
HTTP messages typically contain, **request/response line**, **request/response headers**, and/or **request/response body**

HTTP Messages

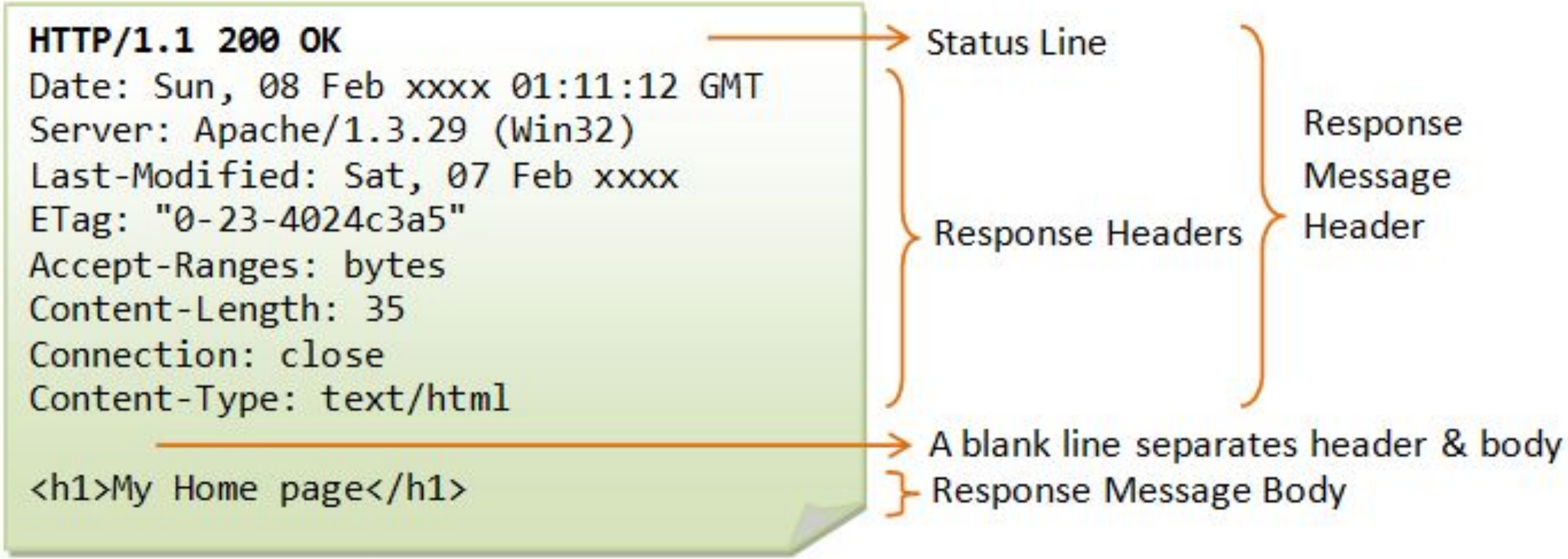
HTTP/1.X vs HTTP/2



HTTP Request Message



HTTP Response Message



HTTP Request Methods

GET

The GET method requests a representation of the specified resource. Requests using GET should only retrieve data

POST

The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server

HTTP Request Methods

PUT

The PUT method replaces all current representations of the target resource with the request payload

DELETE

The DELETE method deletes the specified resource.

PATCH

The PATCH method is used to apply partial modifications to a resource

HTTP Request Methods

HEAD

The HEAD method asks for a response identical to that of a GET request, but without the response body

OPTIONS

The OPTIONS method is used to describe the communication options for the target resource.

HTTP Request Methods

CONNECT

The CONNECT method establishes a tunnel to the server identified by the target resource.

TRACE

The TRACE method performs a message loop-back test along the path to the target resource.

`HTTP` `(Un)Safe` Request Methods

A method is considered safe if it **doesn't change the state of the server**

The server provides only information

`GET`, `HEAD`, `OPTIONS`, and `TRACE` are safe methods

`POST`, `PUT`, and `DELETE` methods are not safe as they change the state of the server

`HTTP` `Idempotent` Request Methods

A method is considered idempotent if **the state of the server doesn't change the second time the method is called with the same data**

Safe methods by definition **are** considered **idempotent**

`PUT` and **`DELETE`** are idempotent but not safe

`POST` is neither a safe nor an idempotent method

HTTP **Response Status Codes**

`1xx series` – Informational Message

`2xx` – Success Message

`3xx` – Redirection Message

4xx – Error Messages Related to Client

5xx – Error Messages Related to Server

Readings

The WebSocket Protocol