

# Evolution of HTTP

---

17<sup>th</sup> October 2024

Phillip Benachour

Slides originally developed by Mathew Broadbent. Revised and updated by Phillip Benachour.

# Lecture Structure

---

- What is the *Hyper Text Transfer Protocol* ?
- Why is it so important?
- HTTP/0.9 & HTTP/1.0
- HTTP/1.1
- HTTP/2
- HTTP/3 & QUIC



# What is the *Hyper Text Transfer Protocol* ?

---

# What is HTTP?

---

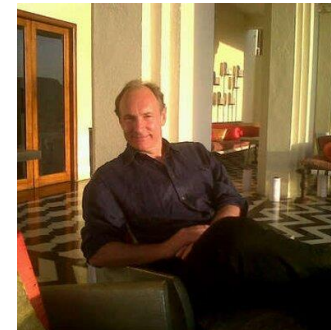
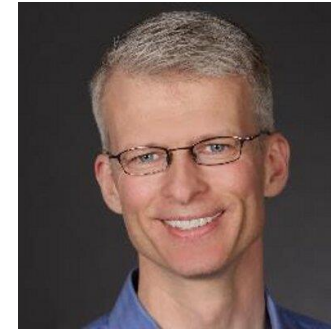
- *Application Layer Protocol*
  - *Sits on top of TCP (currently!)*
- *Used primarily on the Web*
  - *Probably for more than you might think!*



# HTTP Origins & History

---

- HTTP/1.0 itself described in RFC 1945
  - May 1996
- *Roy Fielding*
- *Henrik Frystyk Nielsen*<sup>[1]</sup>
- *Tim Berners-Lee*
  - Often credited with inventing the World Wide Web
  - Implemented the first interaction between a *HTTP* client and server over the Internet

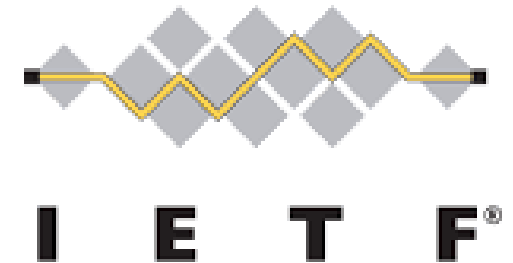


[1] <https://hanselminutes.com/292/history-of-http-and-the-world-wide-web-with-henrik-frystyk-nielsen>

# HTTP Ownership

---

- World Wide Web Consortium (W3C)
  - Main standards organisation for the world wide web
  - Led by Tim Berners-Lee
  - Develops presentation and data representations: HTML, XML
- Internet Engineering Task Force (IETF)
  - Main standards organisation for the Internet
  - Develops transport-layer and application layer protocols: TCP, HTTP



# Web and HTTP

---

- First, reminder...
  - *Web page* consists of *objects*
    - Object can be HTML file, JPEG image, Java applet, audio file,...
  - Web page consists of *base HTML-file* which includes *several referenced objects*
    - Each object is addressable by a *URL*, e.g.,

www.someschool.edu/someDept/pic.gif

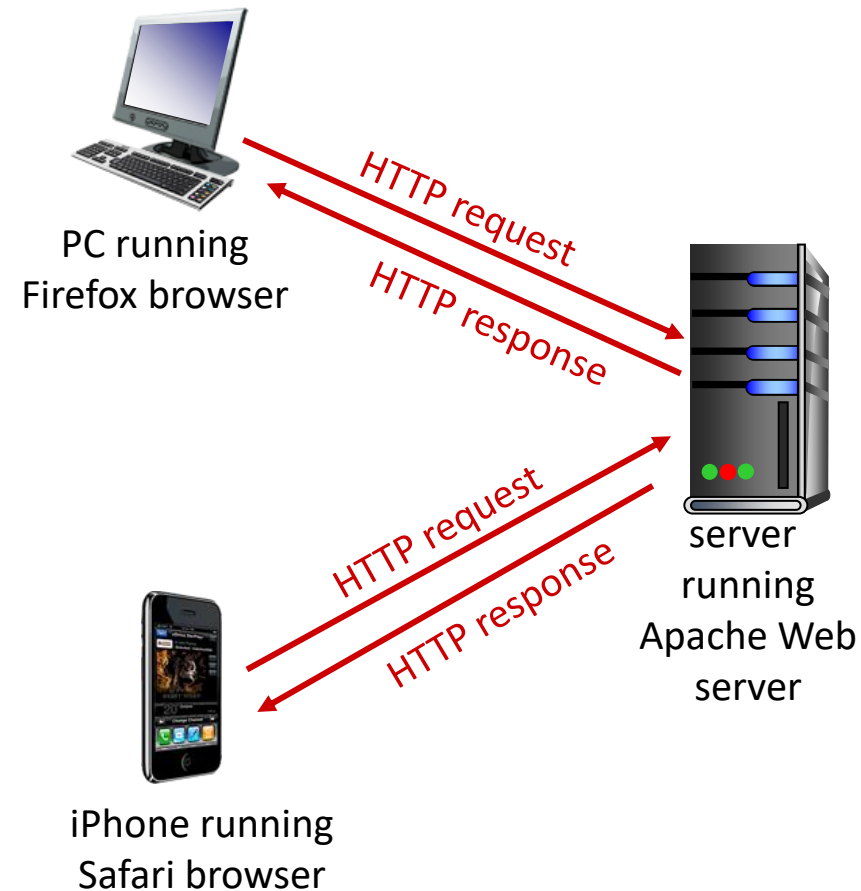
host name

path name

# HTTP Overview

---

- Web's application layer protocol
- Client/server model
  - *Client*: browser that requests, receives, (using HTTP protocol) and “displays” Web objects
  - *Server*: Web server sends (using HTTP protocol) objects in response to requests





# HTTP Overview (continued)

---

- Underlying protocol: TCP
  - Client initiates TCP connection (creates socket) to server, port 80
  - Server accepts TCP connection from client
  - HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
  - TCP connection closed

# Why is HTTP so important?

---

# Importance of HTTP

---

- HTTP underpins many of the transactions that take place on the web
- Most (if not all) of you use it everyday
- From:
  - delivery of websites
  - transactions within applications
  - sharing of multimedia: video, audio
- Very versatile and has evolved significantly over time
- The most influential protocol from a global perspective?



# HTTP Messages and Formats

---

# HTTP request message

- Two types of HTTP messages: *request, response*
- **HTTP request message:**
  - ASCII (human-readable format)

request line  
(GET, POST,  
HEAD commands)

header  
lines

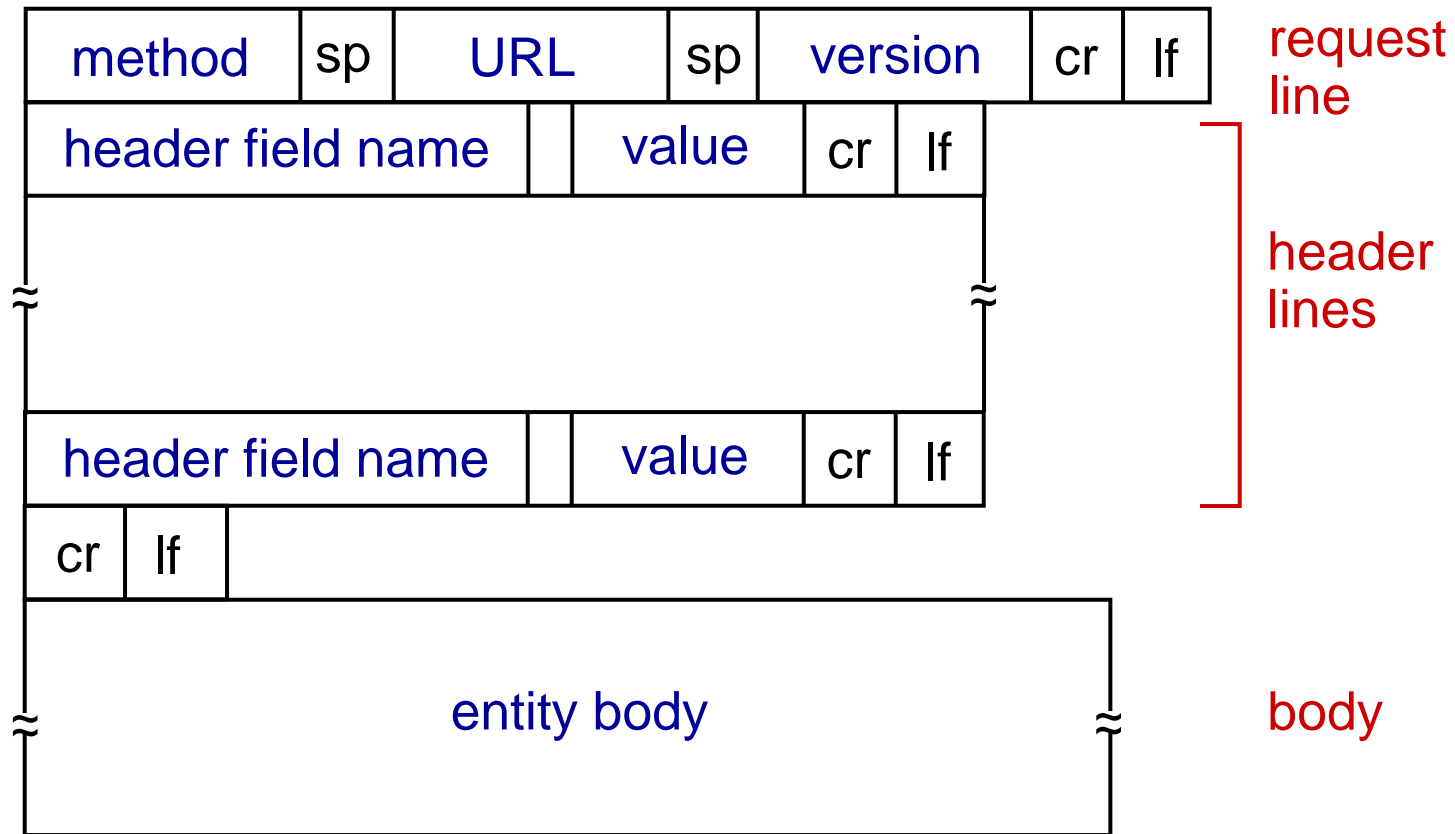
carriage return,  
line feed at start  
of line indicates  
end of header lines

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

carriage return character

line-feed character

# HTTP request message: general format



# HTTP response message

status line  
(protocol  
status code  
status phrase)

header  
lines

data, e.g.,  
requested  
HTML file

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02
GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-
1\r\n
\r\n
data data data data data ...
```

# HTTP response status codes

- Status code appears in 1st line in server-to-client response message.
- Some sample codes:

## **200 OK**

- Request succeeded, requested object later in this msg

## **301 Moved Permanently**

- Requested object moved, new location specified later in this msg (Location:)

## **400 Bad Request**

- Request msg not understood by server

## **404 Not Found**

- Requested document not found on this server

## **505 HTTP Version Not Supported**



# HTTP/0.9 & HTTP/1.0

---

# HTTP/0.9

The one-line protocol

---

- Strictly speaking, the first version of HTTP had no version number
- It has since been called HTTP/0.9 to differentiate it from other versions
- Extremely simple!
- Requests consist of a single line
- Only GET method available

# HTTP/0.9

The one-line protocol

---

- *Request:*

```
GET /index.html
```
- *Response:*

```
<HTML>  
Hello world!  
</HTML>
```
- **No URL**
  - Only used through `telnet`, so no protocol, server or port – already connected!
- **No HTTP headers**
  - Only HTML could be transmitted
- **No status or error codes**
  - A specific (human-readable) HTML file sent back instead

# HTTP/1.0

Building functionality further

---

- Over the next few years, use of HTTP grew. By 1995 there were over 18,000 servers handling HTTP traffic on port 80 across the world.
- The protocol had evolved past its HTTP/0.9 roots, and in 1996 RFC 1945 codified HTTP/1.0.
- HTTP/1.0 brought more functionality:
  - Whereas the HTTP/0.9 spec was about a page long, the 1.0 RFC had 60 pages.
- It brought in ideas that are very familiar to us today: Headers, Response codes, Redirects, Content encoding, (compression).

# HTTP/1.0

Building functionality further

---

- With more traction came more functionality
- Versioning in the request (HTTP/0.9 or HTTP/1.0)
- A machine-readable status code
- Importantly: introduced HTTP headers
  - HTTP header fields add additional information to request and response messages.
  - Metadata could be exchanged – flexible and extensible for the future
  - A reason for its continued success?
- Using these headers, documents other than HTML can be transmitted
  - Using the `Content-Type` header

# HTTP/1.0

Building functionality further

---

- Headers:
  - HTTP header fields add additional information to request and response messages.
  - Lets the client and the server pass additional information with an HTTP request or response.
  - a field of an HTTP request or response that passes additional context and metadata about the request or response.
    - For example, a **request message** can use headers to indicate it's preferred media formats, while a **response** can use header to indicate the media format of the returned body.

# Header and Methods examples

Header example	Description
Date: Tue, 25 Oct 2022 02:16:03 GMT	The date the server generated the response
Content-length: 15040	The entity body contains 15,040 bytes of data
Content-type: image/gif	The entity body is a GIF image
Accept: image/gif, image/jpeg, text/html	The client accepts GIF and JPEG images and HTML

Method example	Description
GET	Get a document from the server.
HEAD	Get just the headers for a document from the server.
POST	Send data to the server for processing.

# HTTP/1.0

Building functionality further

---

- *Request:* `GET /index.html HTTP/1.0`
- *Response:*

```
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 of a cat
  <IMG SRC="/cat.gif">
</HTML>
```



# HTTP/1.0

Building functionality further

---

- 1991-1995
  - Features were introduced on a trial-and-error basis (!)
  - Different vendors/developers pushed many different functionalities
  - If it gained popularity, it was adopted by others (to keep up!)
  - Interoperability issues common 😞
- 1996
  - To bring all parties into a common agreement, an RFC was developed and agreed upon (see previous slide)
  - Technically, never actually standardized!

# HTTP/1.0

## Connections

---

- Non-persistent HTTP
  - At most one object sent over TCP connection
    - Connection then closed!
  - Downloading multiple objects required multiple connections

User enters URL: `www.someSchool.edu/someDepartment/home.index`

(contains text,  
references to 10  
jpeg images)

1a. HTTP client initiates TCP connection to HTTP server (process) at `www.someSchool.edu` on port 80

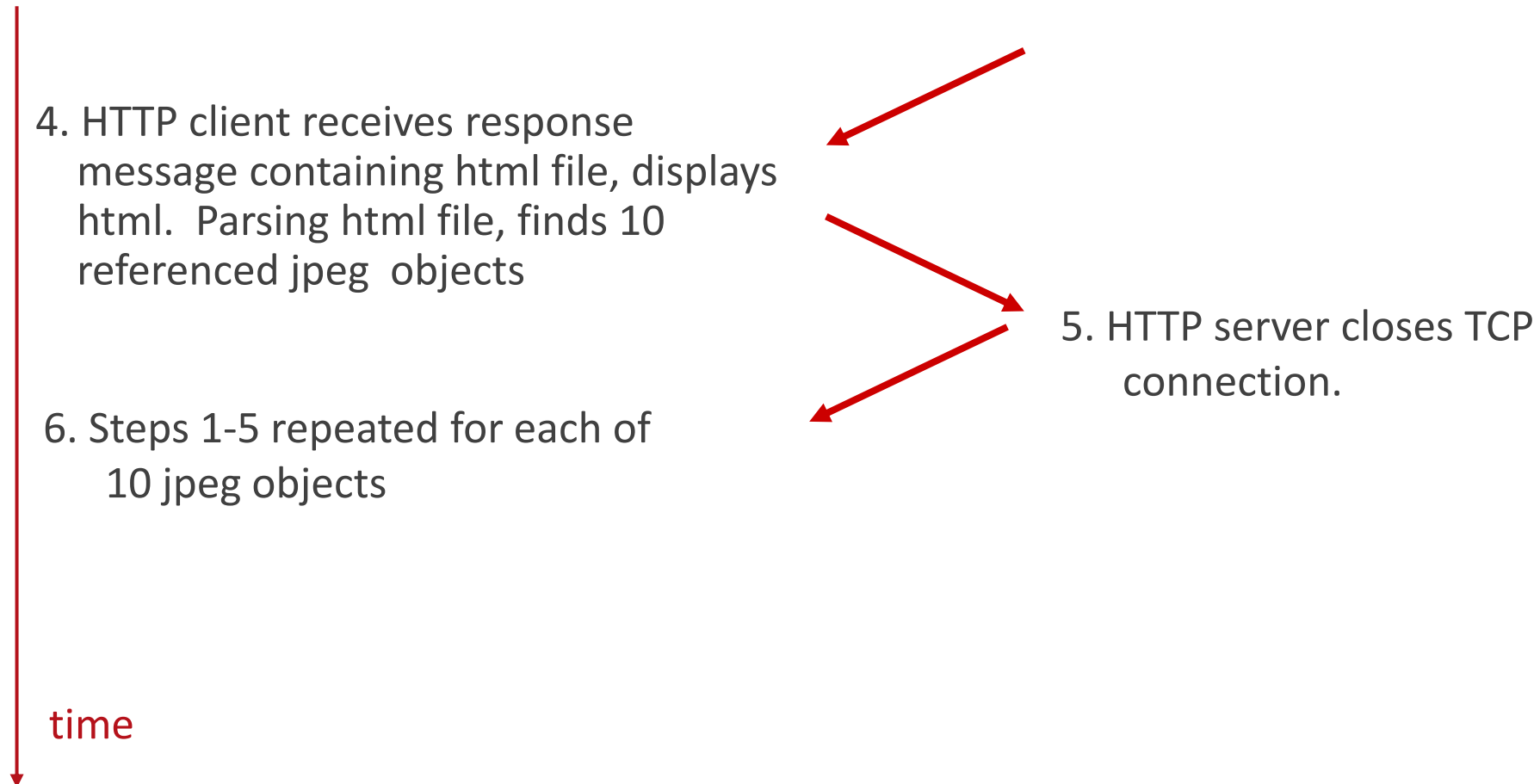
1b. HTTP server at host `www.someSchool.edu` waiting for TCP connection at port 80. “accepts” connection, notifying client

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object `someDepartment/home.index`

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

time

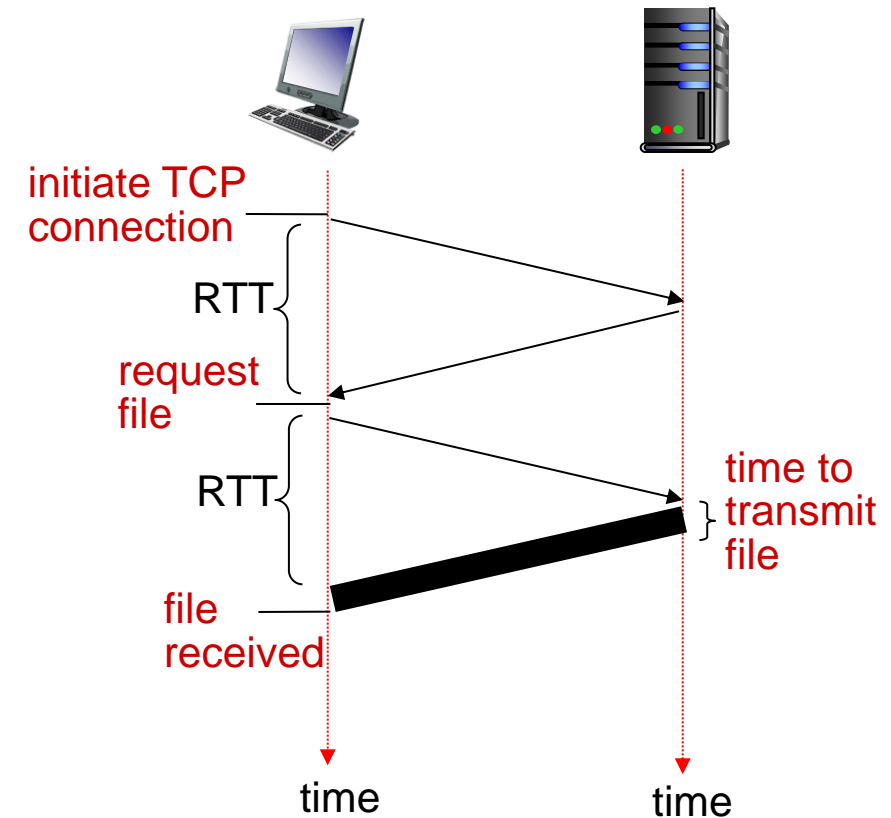




# Non-persistent HTTP: response time

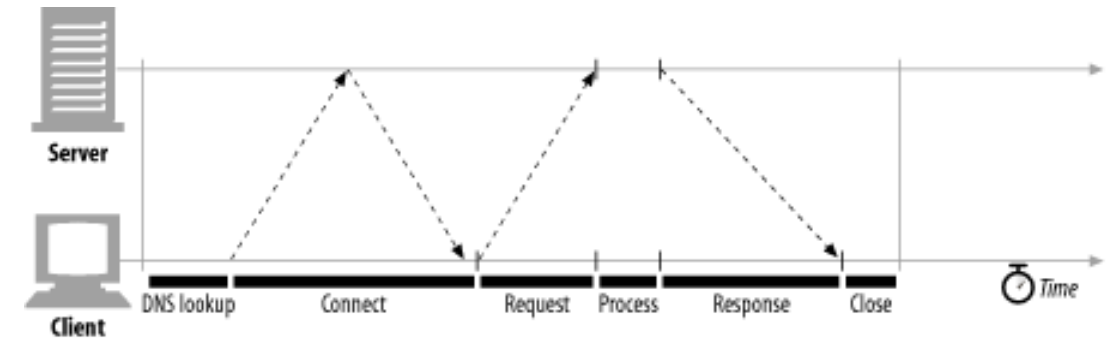
- *RTT (definition)*: time for a small packet to travel from client to server and back
- *HTTP response time*:
  - One RTT to initiate TCP connection
  - One RTT for HTTP request and first few bytes of HTTP response to return
  - File transmission time
  - Non-persistent HTTP response

$$\text{time} = 2\text{RTT} + \text{file transmission time}$$



# TCP Performance Considerations

- HTTP is layered directly on TCP.
- Performance of HTTP transactions depends critically on the performance of the underlying TCP.
- Transaction processing time can sometimes **be quite small** compared to the time required to set up TCP connections and transfer the request and response messages.
- Unless the client or server is overloaded, most HTTP delays are caused by TCP network delays.



# HTTP/1.0

## State

---

- Fundamentally, the server holds no state for each client
  - Not part of the definition in the RFC
- Stateless!
- ~1996: WebDAV Standard
  - Treat the web as a distributed file system
  - Further extended for specific applications
    - CardDAV for address book
    - CalDav for calendars
  - Interoperable
  - Issue: \*DAV extensions had to be implemented by the servers – complex!

# Recap: Comparing HTTP

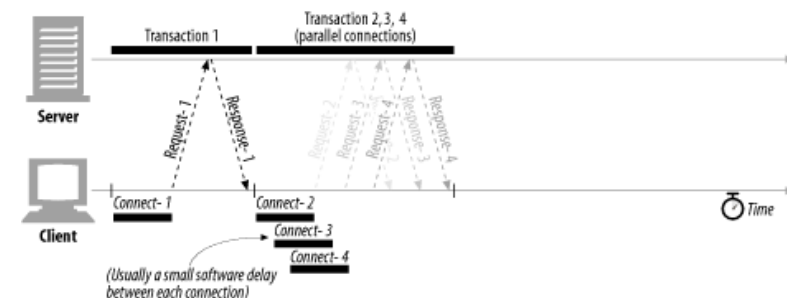
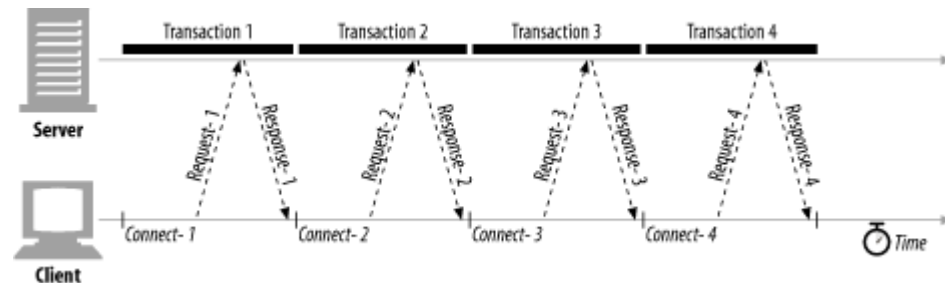
---

- HTTP/0.9
  - Non-persistent
  - Stateless
  - Methods: GET
- HTTP/1.0
  - Non-persistent
  - Stateless
  - Methods: GET, HEAD, POST



# Issues

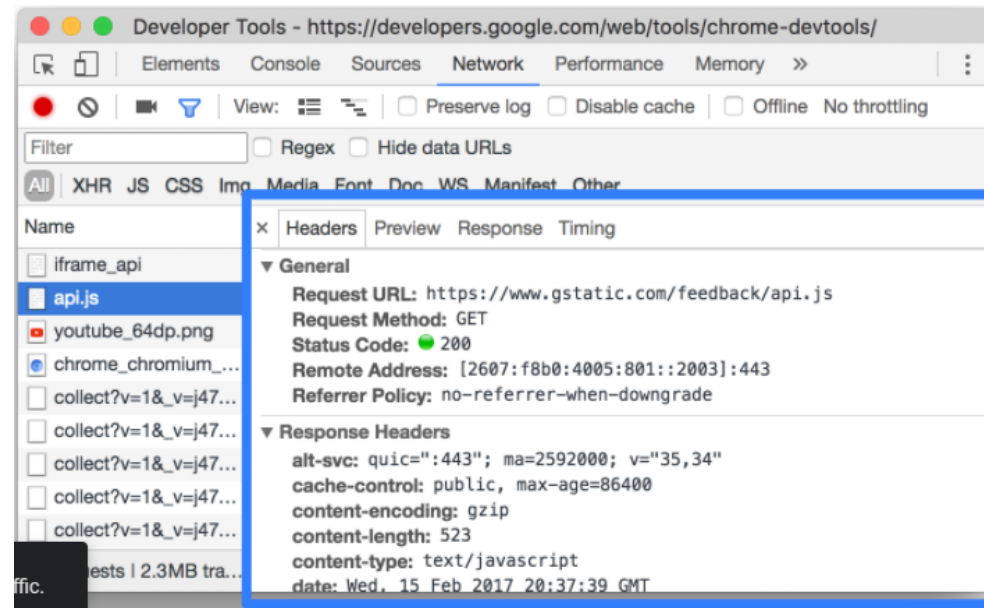
- Non-persistent HTTP limitations
  - Requires 2 RTTs per object
  - Operating System overhead for each TCP connection
  - Browsers would often circumvent this by opening multiple parallel TCP connections to fetch reference objects.
    - Parallel connections may make pages load faster!



# Task 1

## Investigating HTTP Headers with Chrome

- Open up your browser and load developer tools
- To view HTTP header data about a request:
  - Click on the URL of the request, under the **Name** column of the Requests table
  - Click the **Headers** tab





# HTTP/1.1

---

# HTTP/1.1

## Standardized HTTP

---

- Before even the publication of the HTTP/1.0 document, work was ongoing to (officially) standardize protocol
- 1997 (a few months after HTTP/1.0!)
  - HTTP/1.1 becomes the first official standard
- It remained the de-facto standard used on the Internet today
  - Refined many times since then in additional RFCs
  - Most recent being 2014 (!)

# HTTP/1.1

Standardised HTTP

- 
- Methods
  - Connection reuse
    - Persistent HTTP!
  - Pipelining
    - Sending the next request before the first is fully transmitted
  - Host header
    - Previously, a single server could only host a single domain-website.
    - With HTTP/1.1 multiple websites running on the same underline server. Enables differentiation.

# HTTP/1.1

## Standardised HTTP

---

- Chunked responses:
  - Sending files back to the client did not need to fit objects into a single HTTP packet.
  - Instead, they could be spread across different packets and the client would re-assemble them. Key for multimedia over the web (large files).
- Cache control mechanisms:
  - Copies of your content can be located close to your location.
- Content metadata
  - Language, encoding, etc.

# HTTP/1.1

## Methods Headers comparison HTTP/1.0 v HTTP/1.1

Method	Description	HTTP/1.0	HTTP/1.1
GET	Retrieve the information specified.		
HEAD	Identical to the GET request, but the server must not return any page content other than the HTTP headers.		
POST	Allows the client to submit information to the server, used for submitting information from a form for example.		
PUT	Allows the client to place an item on the server in the location specified.		
DELETE	Allows the client to delete the item specified in the request.		
TRACE	Allows the client to see the request it made to the server. This acts as a loopback in effect.		
OPTIONS	Allows the client to determine the communications options available on the server.		

# HTTP/1.1

## Connections

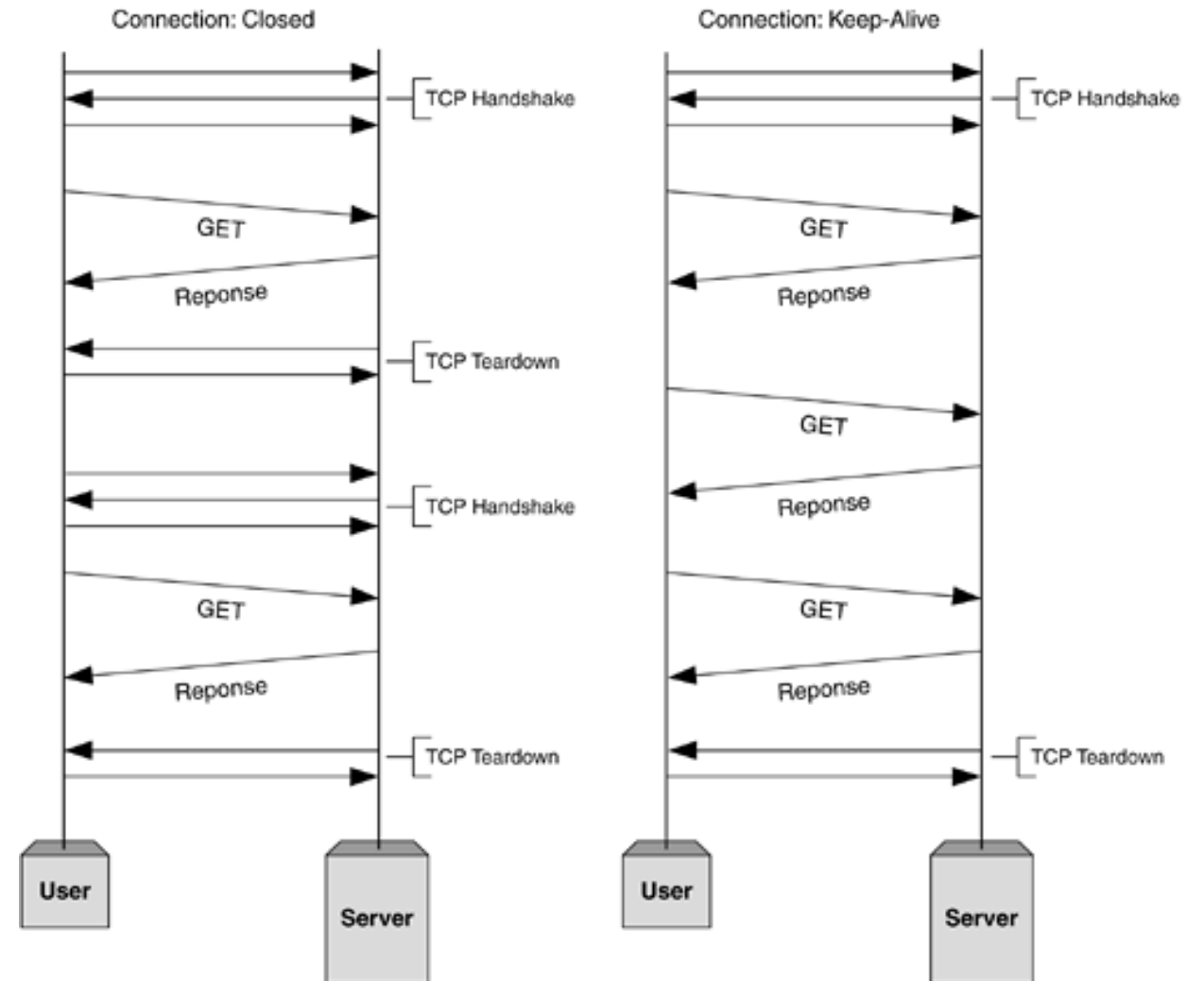
---

- Persistent connection in HTTP!
- Multiple objects can be sent over single TCP connection between client, server
  - Server leaves connection open after sending response
  - Subsequent HTTP messages between same client/server sent over open connection
  - Client sends requests as soon as it encounters a referenced object
    - As little as one RTT for all the referenced objects
- Enables pipelining: a subsequent request sent before the first is resolved!



# HTTP/1.1

TCP handling between HTTP/1.0 and HTTP/1.1.



In a default HTTP/1.0 session, the TCP connection will be torn down and re-established between each HTTP GET request.

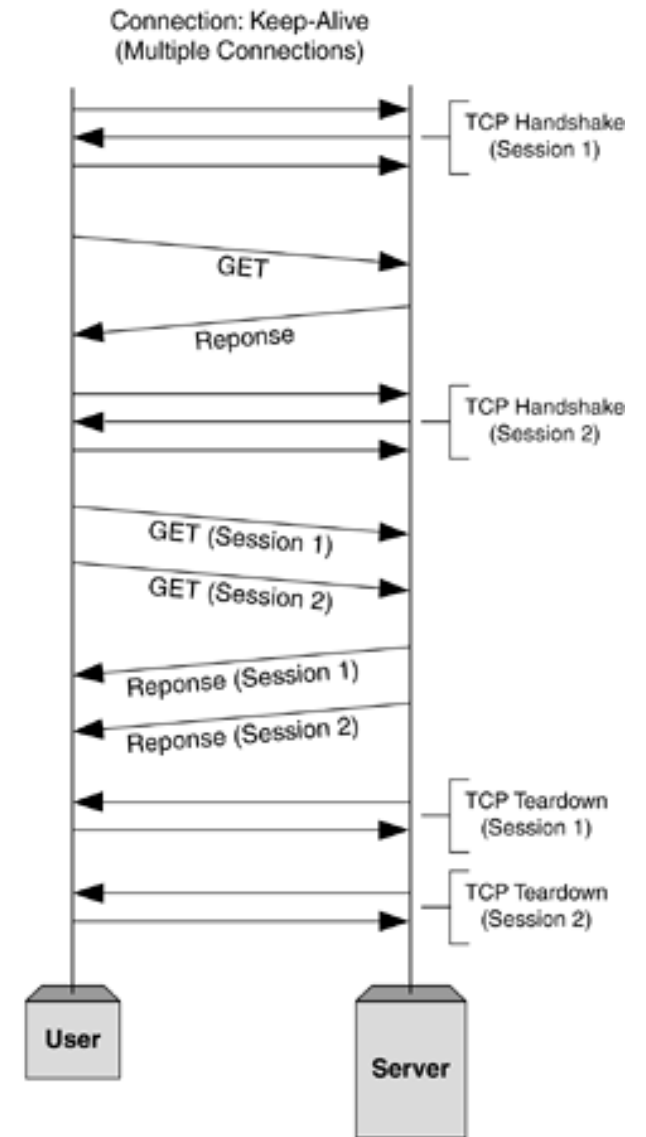
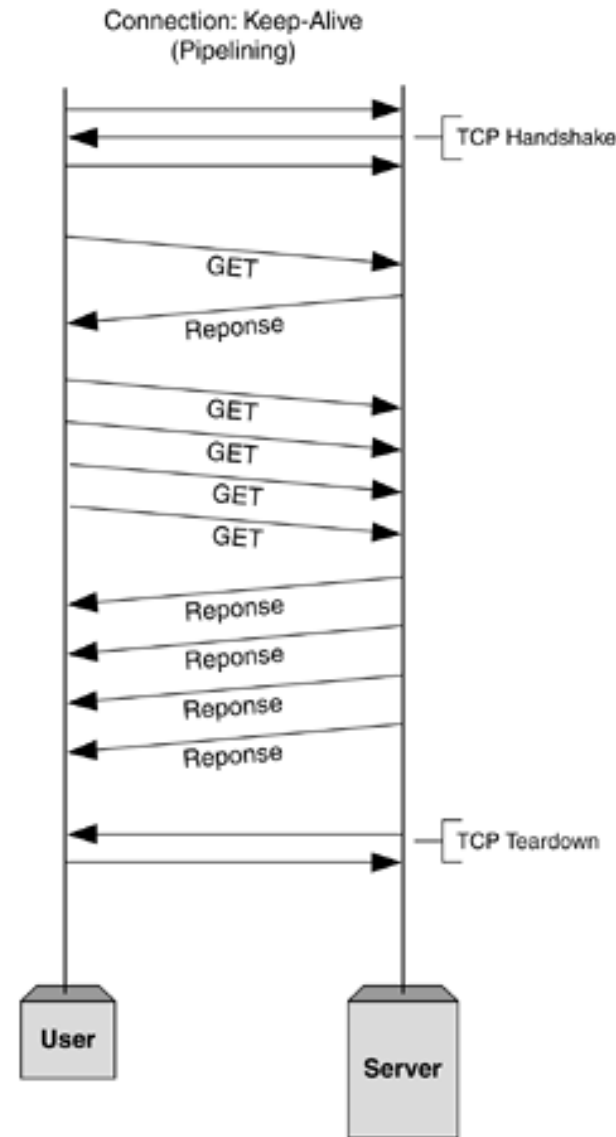
In a default HTTP/1.1 session, a single TCP connection will be held, open and multiple GET requests will be passed across.

# HTTP/1.1

## Pipelining vs. Multiple

Implementing pipelining and multiple connections as performance mechanisms.

- Pipelining is a feature that allows a client to send all of its requests at once.
- There were a couple of problems with pipelining that prevented its popularity:
  - Servers still had to respond to the requests in order.
  - This meant if one request takes a long time, this **head of line blocking** will get in the way of the other requests.



# HTTP/1.1

## Connection Management

---

- Keep-Alive
  - Used prior to 1.1 and obsoleted as it became the default
  - Now utilised to define policies
  - Can be used by client, server, proxies

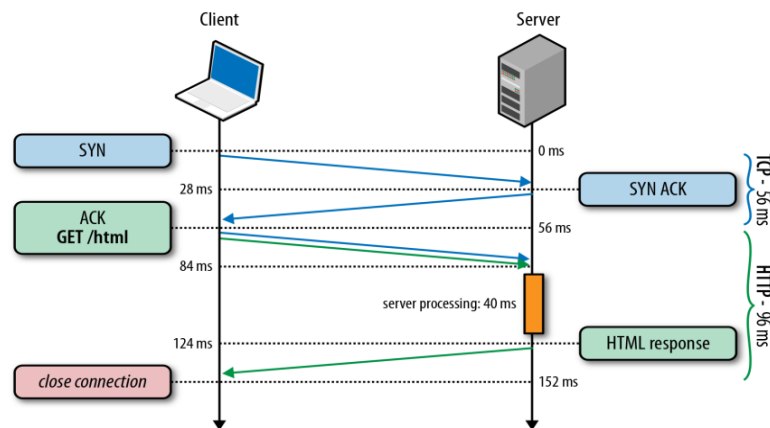
```
Keep-Alive: timeout=5, max=1000
```

- Upgrade
  - Allows an initial connection with a common protocol
  - Subsequent upgrade and flexibility to do so.

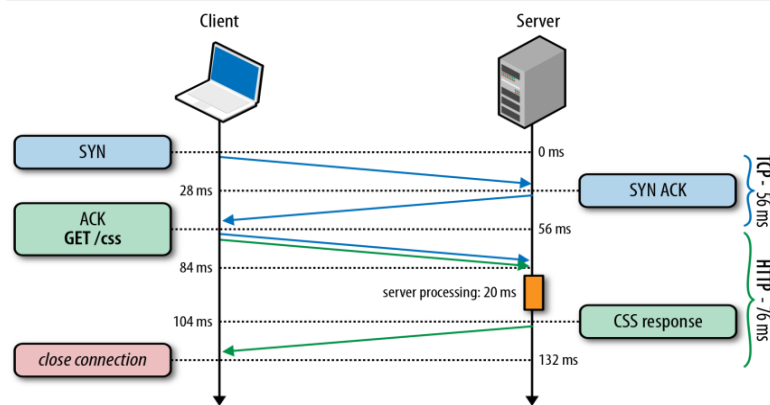
```
Upgrade: HTTP/2.0
```

# Benefits of Keepalive Connections

TCP connection #1, Request #1: HTML request

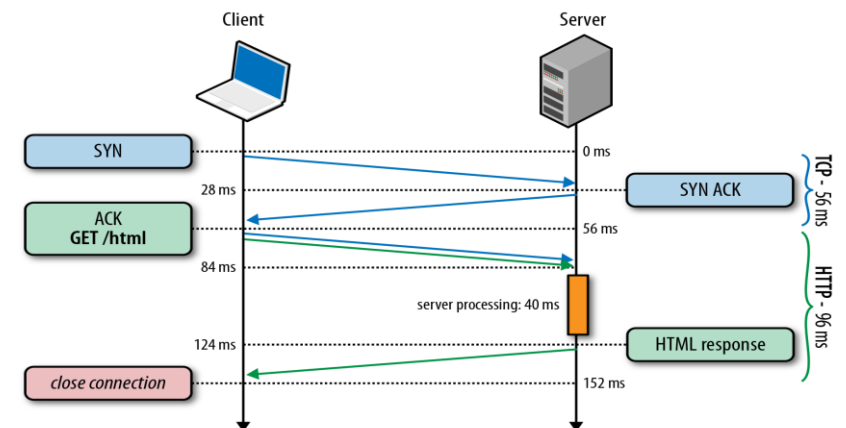


TCP connection #2, Request #2: CSS request

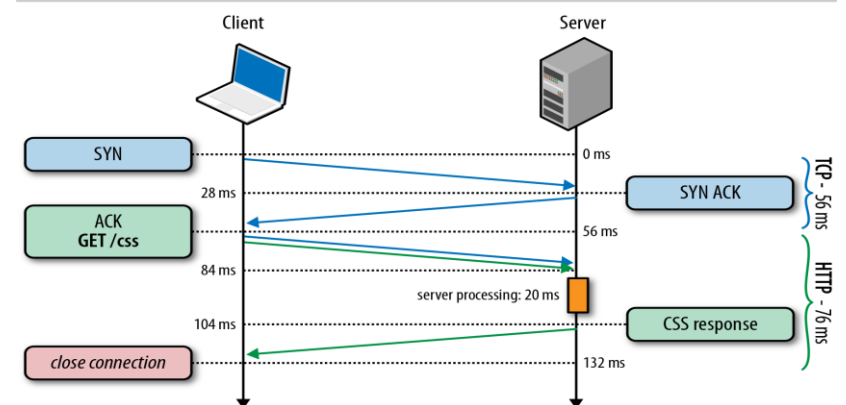


- TCP connection: assume 28 ms one-way “light in fibre” delay between NYK and London.
- Fetching two small resources (<4 KB of data): HTML, CSS files.
- The server processing time will vary depending on the resource and the back-end.
  - The faster the processing time the higher the impact of the fixed latency.
- Minimum total time for an HTTP request via TCP is 2 x network round trips.

TCP connection #1, Request #1: HTML request

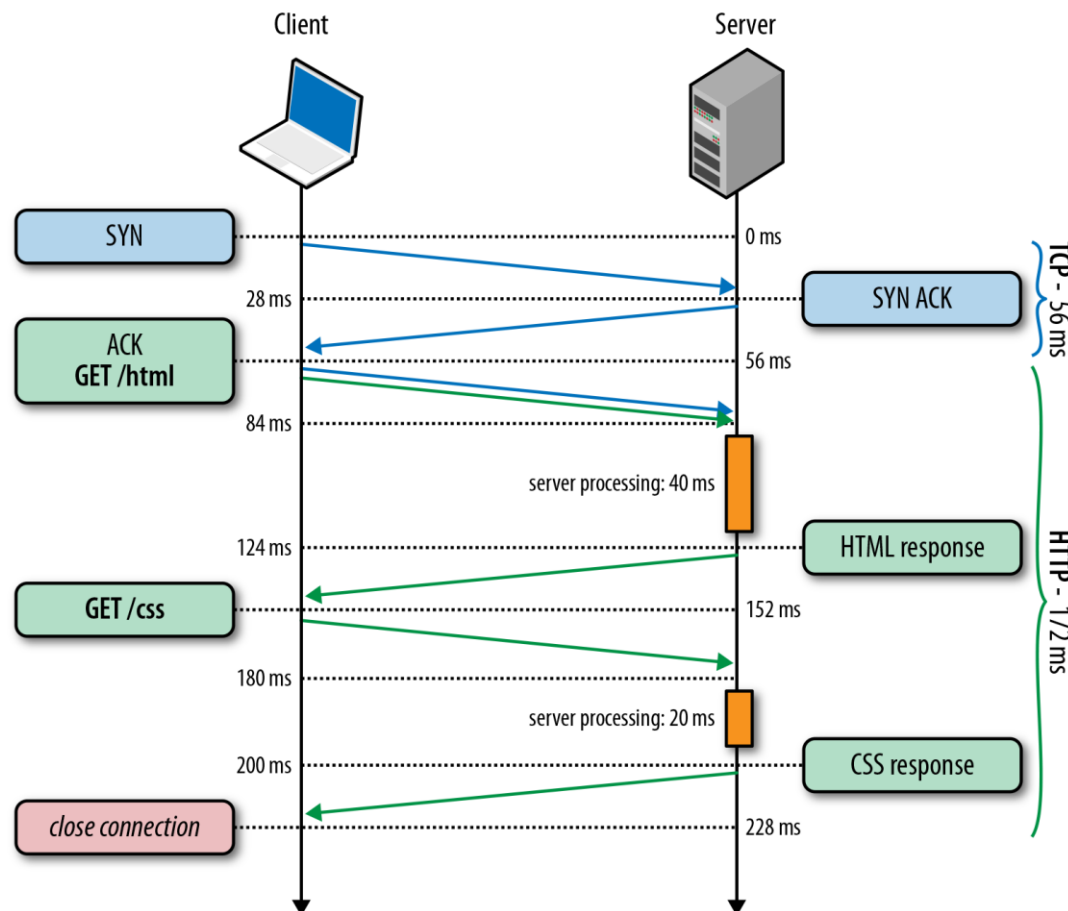


TCP connection #2, Request #2: CSS request



# Benefits of Keepalive Connections

TCP connection #1, Request #1-2: HTML + CSS



- We can optimise using one TCP connection and add support for HTTP Keepalive.
- With two requests, the total savings is a single roundtrip of latency.
- If we consider the general case with a single TCP connection and “M” HTTP requests:
  - No-keepalive, request will incur two roundtrips of latency.
  - With keepalive, the first request incurs two roundtrips, and all following requests incur just one roundtrip of latency.
- Total latency savings for M requests is  $(M-1) \times \text{RTT}$  when connection keepalive is enabled.

# HTTP/1.1

Host Headers: multiple websites at the same domain

- One of the drawbacks of HTTP/1.0 is that a user's request would not contain the host element e.g. [www.bbc.co.uk](http://www.bbc.co.uk), in the GET request to the server.
- This set a limitation if virtual hosting is used at the server side.
  - Where the server is hosting multiple Web sites.
  - Where the server needs to determine the path and page a user is requesting.
- Browser support for the “Host” Header, enables:
  - Users’ requested URL to be converted to a GET request.
  - The request then contains the full path name (file) && host address from which the content is being requested (fetched).

# HTTP/1.1

## Standardized HTTP

---

- *Request:*

```
GET /welcome HTTP/1.1
Host: www.lancaster.ac.uk
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
10.9; rv:50.0) Gecko/20100101 Firefox/50.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.
9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://lancaster.ac.uk/
```

# HTTP/1.1

## Standardized HTTP

---

- *Response:*

```
200 OK
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Date: Mon, 10 Dec 2020 10:55:30 GMT
Keep-Alive: timeout=5, max=1000
Last-Modified: Tue, 19 Sep 2020 00:59:33 GMT
Server: Apache
Transfer-Encoding: chunked
Vary: Cookie, Accept-Encoding.0
```



# HTTP/1.1

15 years of extensions of use until HTTP/2

---

- Where do we go from here?
- HTTP allows for extensibility
  - New headers and methods are easy to add
- This allows it evolve without the need to revise the underlying specification

# HTTP/1.1

State

---

- Fundamentally, the server *still* holds no state for each client
- *Still* Stateless!

# HTTP/1.1

## State

---

- Cookies allow for:
  - Session management - logins, shopping baskets
  - Personalization - user preferences, themes
  - Tracking - recording user behaviour



- Set: `Set-Cookie: <cookie-name>=<cookie-value>`

- Send: `Cookie: cookie_flavour=choc_chip`

# HTTP/1.1

## Caching

---

- As caching became more widespread (and needed)
- HTTP/1.1 introduced explicit cache-control mechanisms

- Off:

```
Cache-Control: no-store
```

- Revalidate:

```
Cache-Control: no-cache
```

- Private/public:

```
Cache-Control: private  
Cache-Control: public
```

- Expiration:

```
Cache-Control: max-age=31536000
```

- Validation:

```
Cache-Control: must-revalidate
```

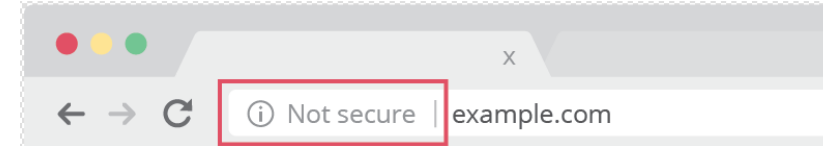
# HTTPS

---

An aside...

# HTTPS

But what version of HTTP is this?!



- HTTP became more prevalent and powerful (banking, shopping, etc.)
- Malicious actors started to emerge and the need to secure communications become important
- HTTP sent over a secure connection - HTTP Secure (HTTPS)
  - Wraps regular HTTP inside a secure 'pipe'
  - Realized initially with SSL, later TLS
- HTTP could be HTTP/1.1, HTTP/2...!
- Can be a source of delay: SSL/TLS handshake can take 1-2 seconds(!)

# Recap: Comparing HTTP

---

- HTTP/0.9
  - Non-persistent connections
  - Stateless
  - Methods: GET
- HTTP/1.0
  - Non-persistent connections
  - Stateless
  - Methods: GET, HEAD, POST
- HTTP/1.1
  - Persistent connections
  - Stateless
  - Methods: GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS

# Issues

---

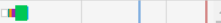

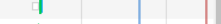

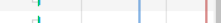
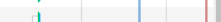

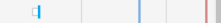

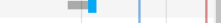
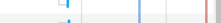
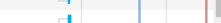
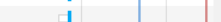

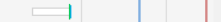



- Pages become more complex and richer
  - More data needs to be transferred!
  - More HTTP requests
  - HTTP/1.1 requests need to be sent in the correct order
  - Several parallel connections possible (~5-8)
    - Complexity and overhead
    - Even with this workaround, *head-of-line blocking* still occurs when these are used up – subsequent requests need to wait for former to complete
- Don't forget: still widely used!



# Task 2

## Investigating HTTP Versions with Chrome

- Open up your browser and load developer tools
- To view HTTP protocol:
  - Right-click on the table headings
  - Select **Protocol**

Name	Status	Protocol	Type	Initiator	Size	Time	Waterfall
www.lancaster.ac.uk	200	http/1.1	document	Other	14.9 kB	123 ms	
app.6.css	200	http/1.1	stylesheet	(index)	(disk cache)	8 ms	
font-face.css	200	http/1.1	stylesheet	(index)	835 B	22 ms	
loadScript.js	200	http/1.1	script	(index)	(disk cache)	5 ms	
jquery-3.5.1.min.js	200	http/1.1	script	(index)	(disk cache)	7 ms	
accessibleMegaMenu.min.js	200	http/1.1	script	(index)	(disk cache)	6 ms	
navigation.bundle.js	200	http/1.1	script	(index)	(disk cache)	5 ms	
scriptHandler.min.js	200	http/1.1	script	(index)	(disk cache)	5 ms	
jquery.unveil.min.js	200	http/1.1	script	(index)	(disk cache)	6 ms	
polyfill.min.js?features=Promise,fetch,Elem...	200	h2	script	(index)	(disk cache)	1 ms	
gtm.js?id=GTM-KHTVZS	200	http/1.1	script	(index):55	1.5 kB	165 ms	
carousel.bundle.js	200	http/1.1	script	(index)	(disk cache)	1 ms	
jquery.sticky.min.js	200	http/1.1	script	(index)	(disk cache)	1 ms	
foundation-6.5.3.min.js	200	http/1.1	script	(index)	(disk cache)	4 ms	
slick-custom.min.js	200	http/1.1	script	(index)	(disk cache)	3 ms	
lazy-load.bundle.js	200	http/1.1	script	(index)	(disk cache)	4 ms	
lu-shield.svg	200	http/1.1	svg+xml	(index)	(disk cache)	4 ms	
lu-logo.svg	200	http/1.1	svg+xml	(index)	(disk cache)	4 ms	



# HTTP/2

---

- 
- ~2010: Google pioneers an alternative method of exchanging data
  - SPDY laid the groundwork for HTTP/2 and was responsible for proving out some of its key features such as multiplexing, framing, and header compression, among others.
  - Increase in responsiveness
  - Reduction in duplicate data
  - Foundation of HTTP/2

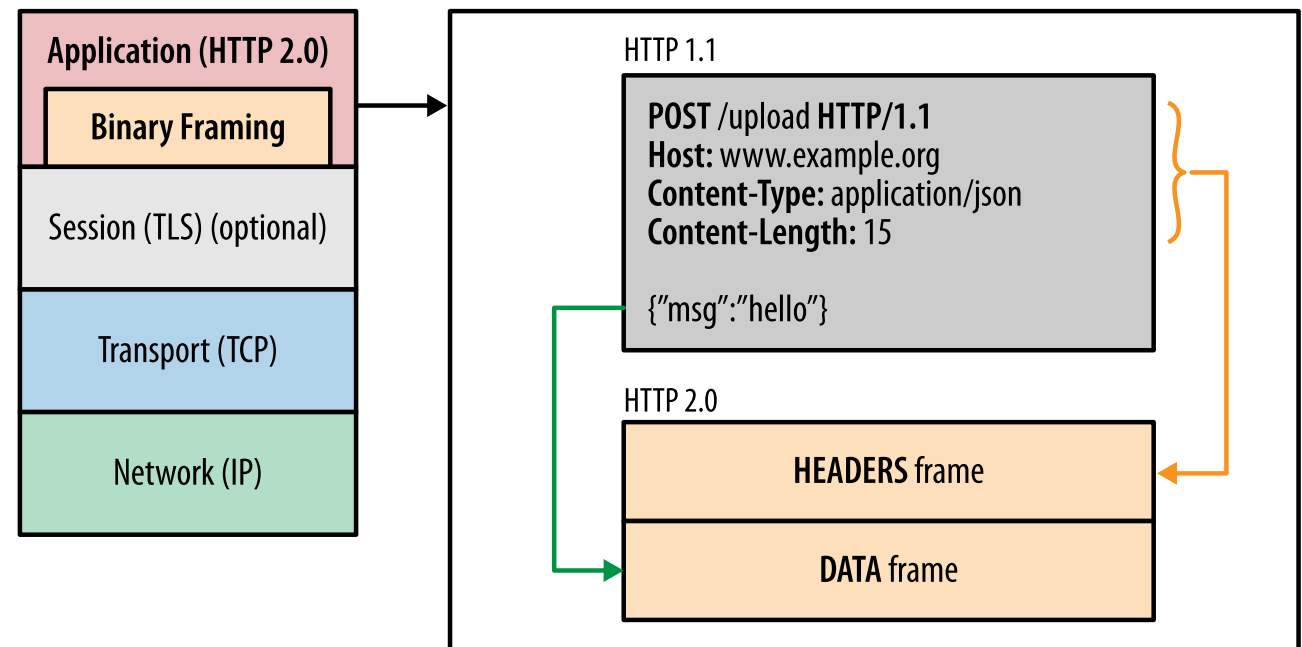
# HTTP/2

The next-generation...

- 
- Binary protocol rather than text
    - Allows further optimization
  - Inherently multiplexed
    - Parallel requests can be handled over the same connection
    - Overcomes the *head-of-line blocking* issue for HTTP
  - Compresses headers
    - Uses inherent commonality between requests to reduce duplication
  - Allows server to populate data in a cache in advance of it being requested:
    - Server push: if you are requesting HTML, you would also get CSS

# Binary framing layer

- The HTTP/2 framing layer is a binary framed protocol. This makes for easy parsing by machines but causes eye strain when read by humans.
- Unlike the newline delimited plaintext HTTP/1.x protocol, all HTTP/2 communication is split into smaller messages and frames, each of which is encoded in binary format.



<https://web.dev/performance-http2/>

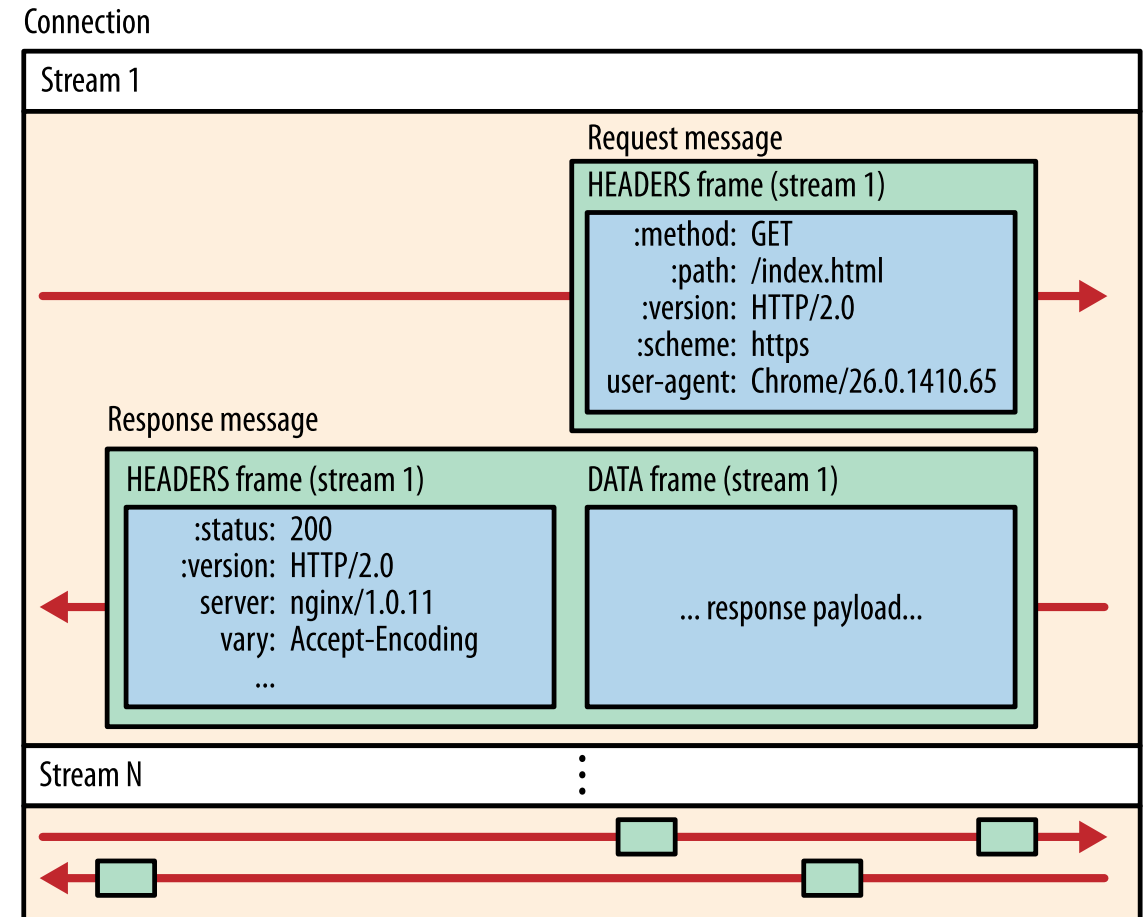
# Frames

---

- HTTP/2 is a framed protocol.
  - Wrapping all the important stuff in a way that makes it easy for consumers of the protocol to read, parse, and create.
- In contrast, HTTP/1 is not framed but is text delimited.

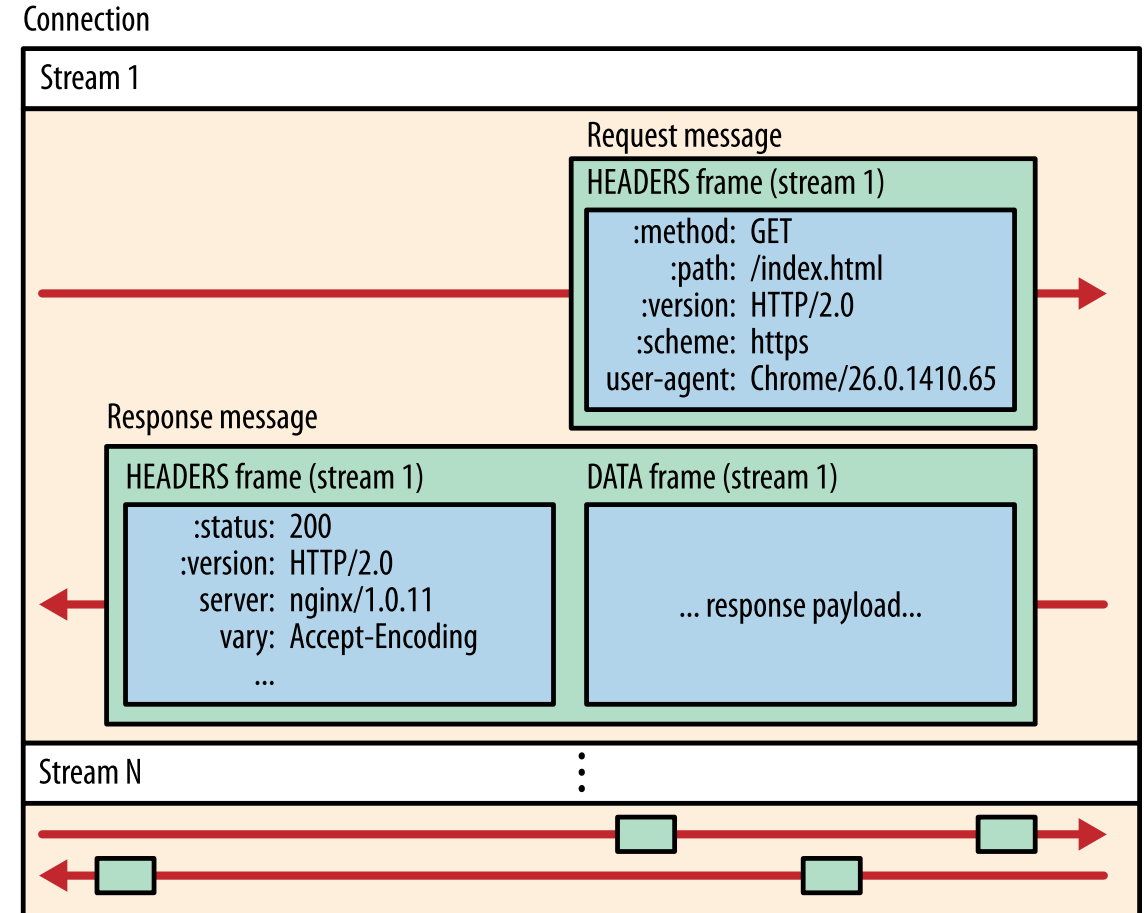
# Streams, messages and frames

- **Stream:** bidirectional flow of bytes (within an established TCP connection), which may carry one or more messages.
- **Message:** complete sequence of frames that are mapped to a logical *request or response message*.
- **Frame:** The smallest unit of communication in HTTP/2, that carries a specific type of data. For example HTTP headers, message payload. Frames from different streams may be interleaved and reassembled.



# Streams, messages and frames

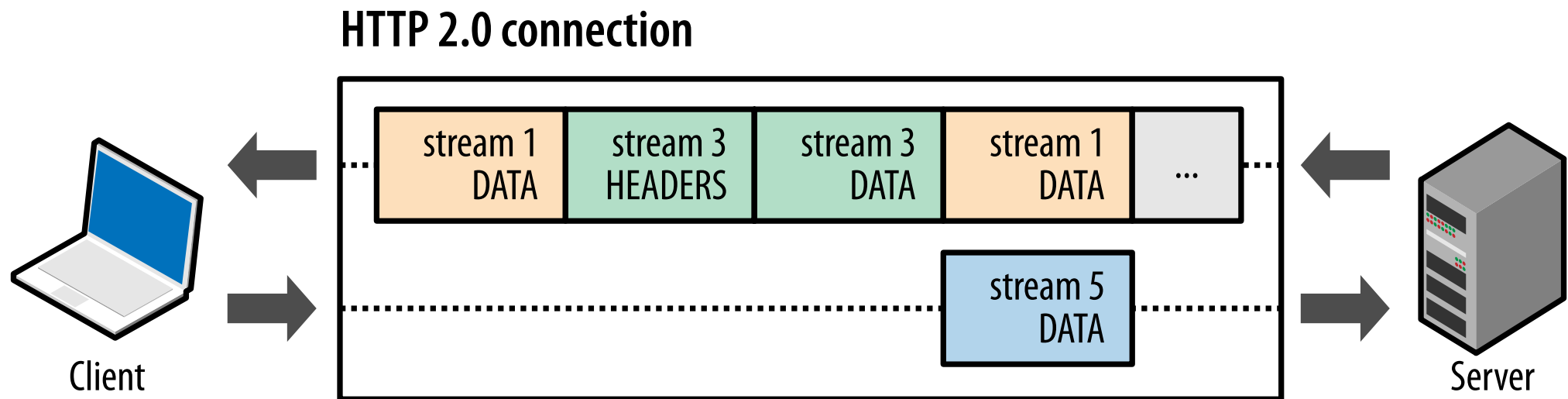
- All communication is performed over a single TCP connection that can carry any number of bidirectional streams.
- Each stream has a unique identifier that is used to carry bidirectional messages.
  - Frames from different streams may be interleaved and then reassembled via the embedded stream identifier in the header of each frame.
- Each message is a logical HTTP message, such as a request, or response, which consists of one or more frames.



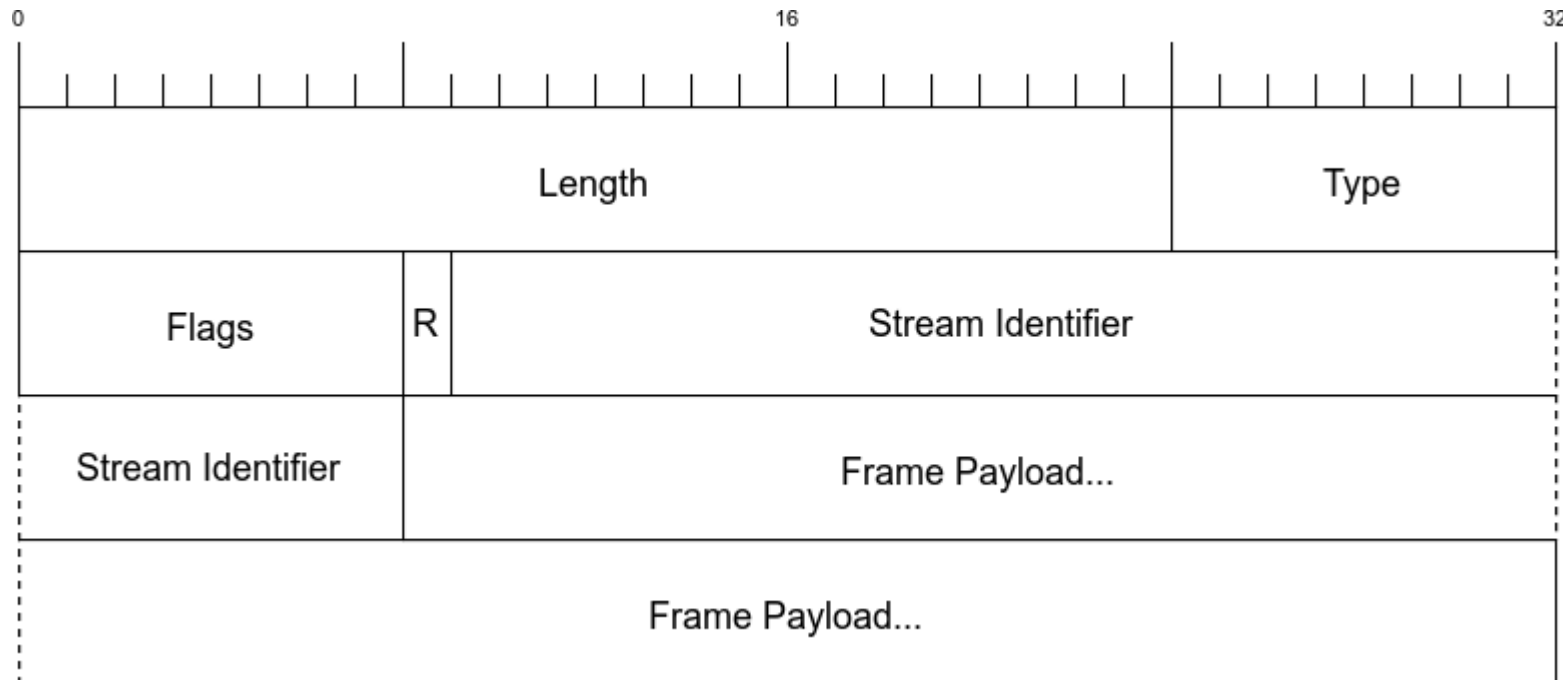


# Request-Response multiplexing

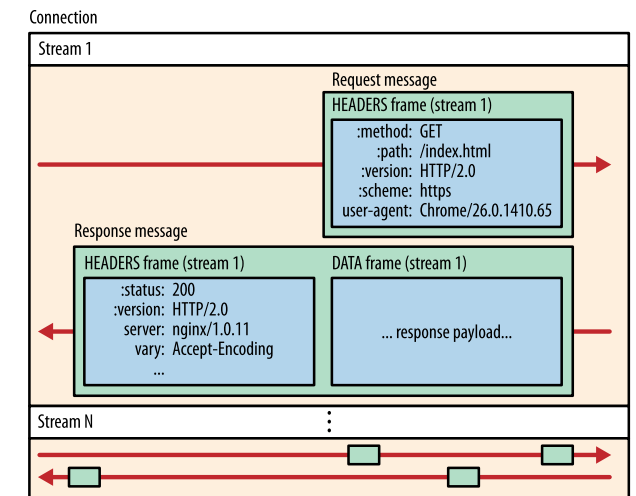
All communication is performed over a single TCP connection that can carry any number of bidirectional streams.



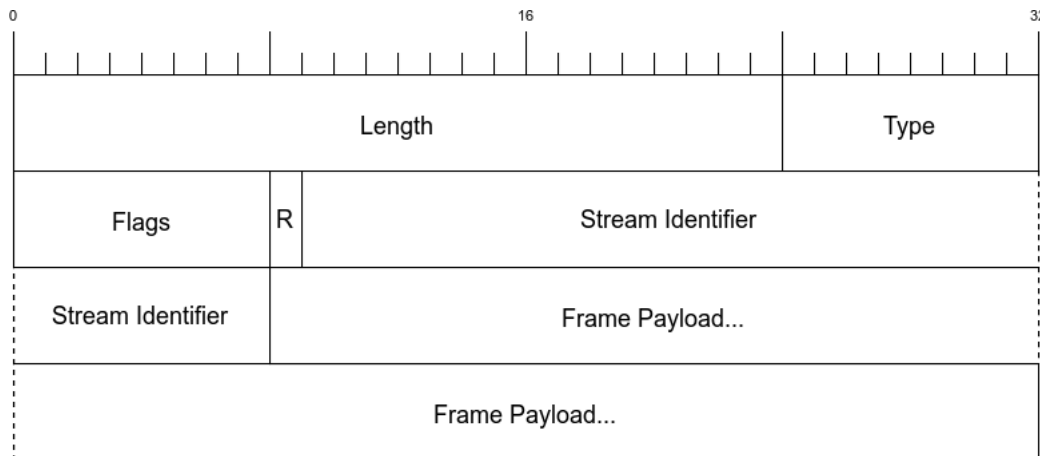
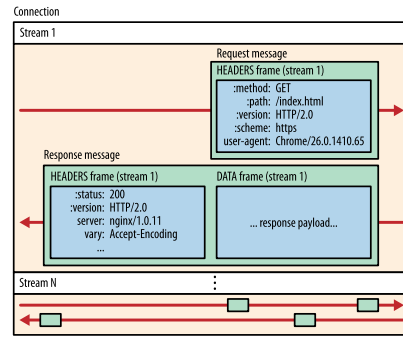
# Frames



HTTP/2 Frame header



# Frames



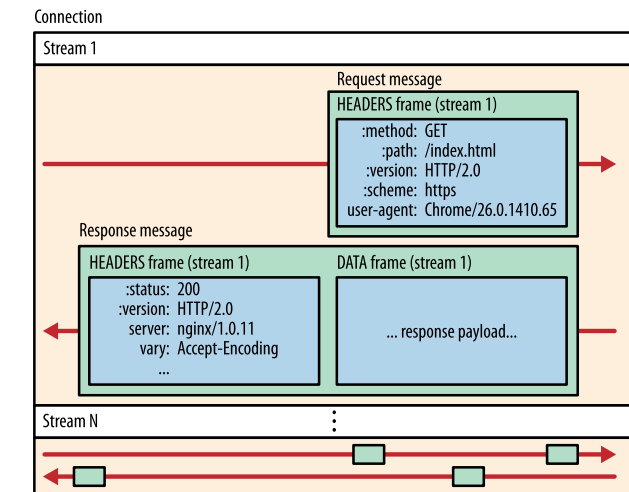
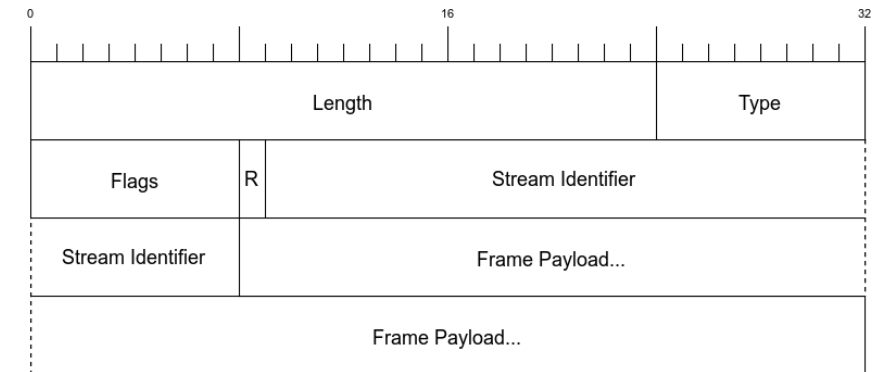
## HTTP/2 frame header

- The first nine bytes (octets) are consistent for every frame.
- The receiver just needs to read those bytes and it knows precisely how many bytes to expect in the whole frame.

Name	Length (bytes)	Description
Length	3	The length of the frame payload (value in the range of $2^{14}$ through $2^{24-1}$ bytes).
Type	1	Type of frame
Flags	1	Flags specific to the frame type.
R	1 (bit)	Reserved bit.
Stream Identifier	31 bits	A unique identifier for each stream.
Frame Payload	Variable	The actual frame content. Its length is indicated in the Length field.

# HTTP/2 Frames Types

Name	Type	Description
Data	0x0	Carries the core content for a stream.
HEADERS	0x1	Contains the HTTP headers and, optionally, priorities
PRIORITY	0x2	Indicates or changes the stream priority and dependencies
....	....	.....
PING	0x6	Tests connectivity and measures round-trip time (RTT)
GOAWAY	0x7	Tells an endpoint that the peer is done accepting new streams



# Streams

## HTTP/2

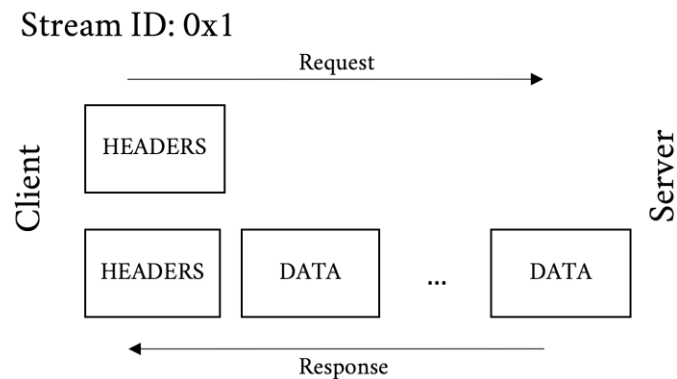
---

- In HTTP/2 a stream is defined as  
“an independent, bidirectional sequence of frames exchanged between the client and server within an HTTP/2 connection.”
- A stream: a series of frames making up an individual HTTP request/response pair on a connection.
  - When a client wants to make a request it initiates a new stream.
  - The server will then reply on that same stream.
- Similar to the request/response flow of HTTP/1
  - Difference: with framing, multiple requests and responses can interleave together without one blocking another. This effect solves the **Head of line blocking issue for HTTP**.
- The Stream Identifier (bytes 6–9 of the frame header) is what indicates which stream a frame belongs to.

# Streams

## HTTP/2

- After a client has established an HTTP/2 connection to the server, it starts a new stream by sending a HEADERS frame.
- A stream is created to transport a pair of request/response messages.
- At a minimum a message consists of a HEADERS frame (which initiates the stream) and can additionally contain DATA frames, as well as additional HEADERS frames.



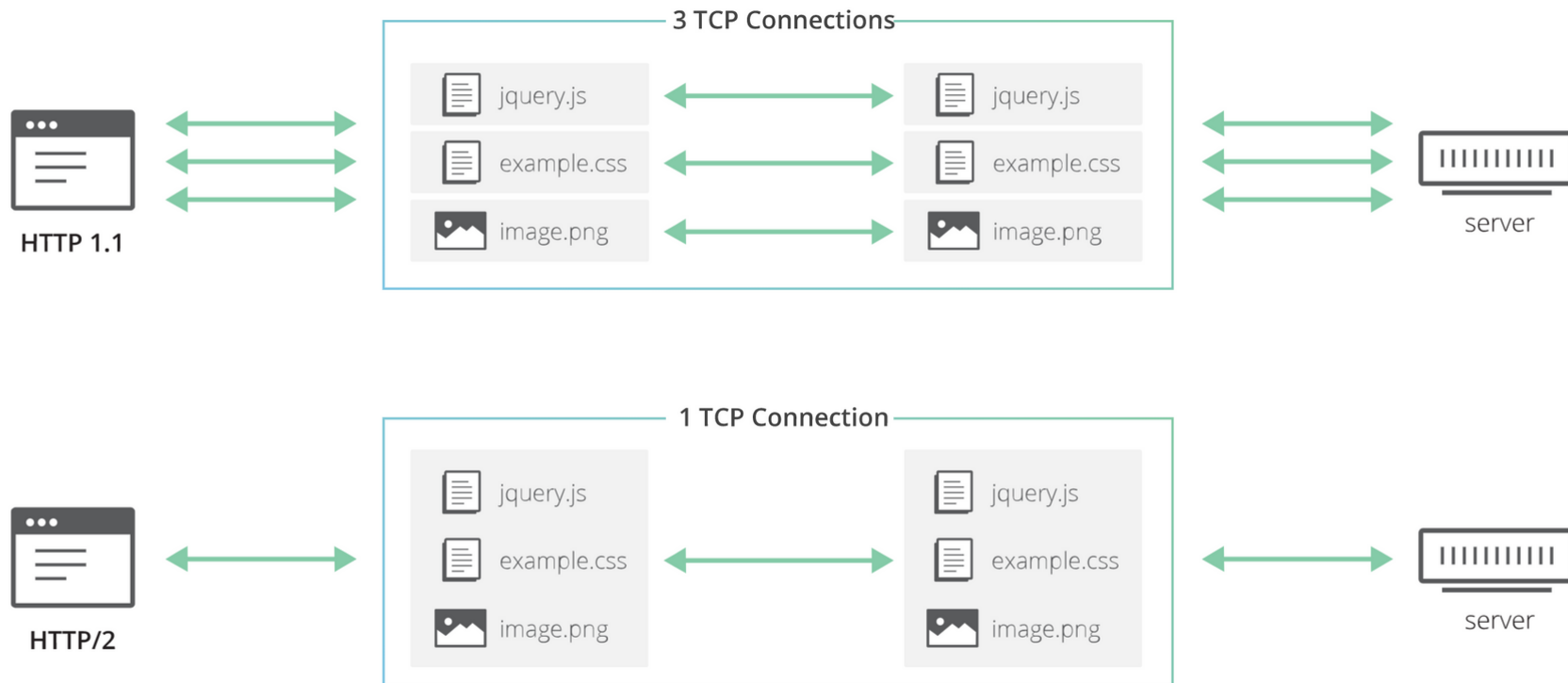
# Connection

---

- The base element of any HTTP/2 session is the connection.
- This is defined as a TCP/IP socket initiated by the client, the entity that will send the HTTP requests.
  - This is no different than the previous HTTP versions;
  - Unlike HTTP/1.1 which is stateless, HTTP/2 bundles connection-level elements that all of the **frames** and **streams** that run over it adhere to.
  - These include connection-level settings and the header table.
  - This implies a certain amount of overhead in each HTTP/2 connection that does not exist in earlier versions of the protocol. The intent is that the benefits of that overhead far outweigh the costs.

# HTTP/2

## Multiple Connections vs. Streams



<https://blog.cloudflare.com/http3-the-past-present-and-future/>



# HTTP/2

The next-generation...

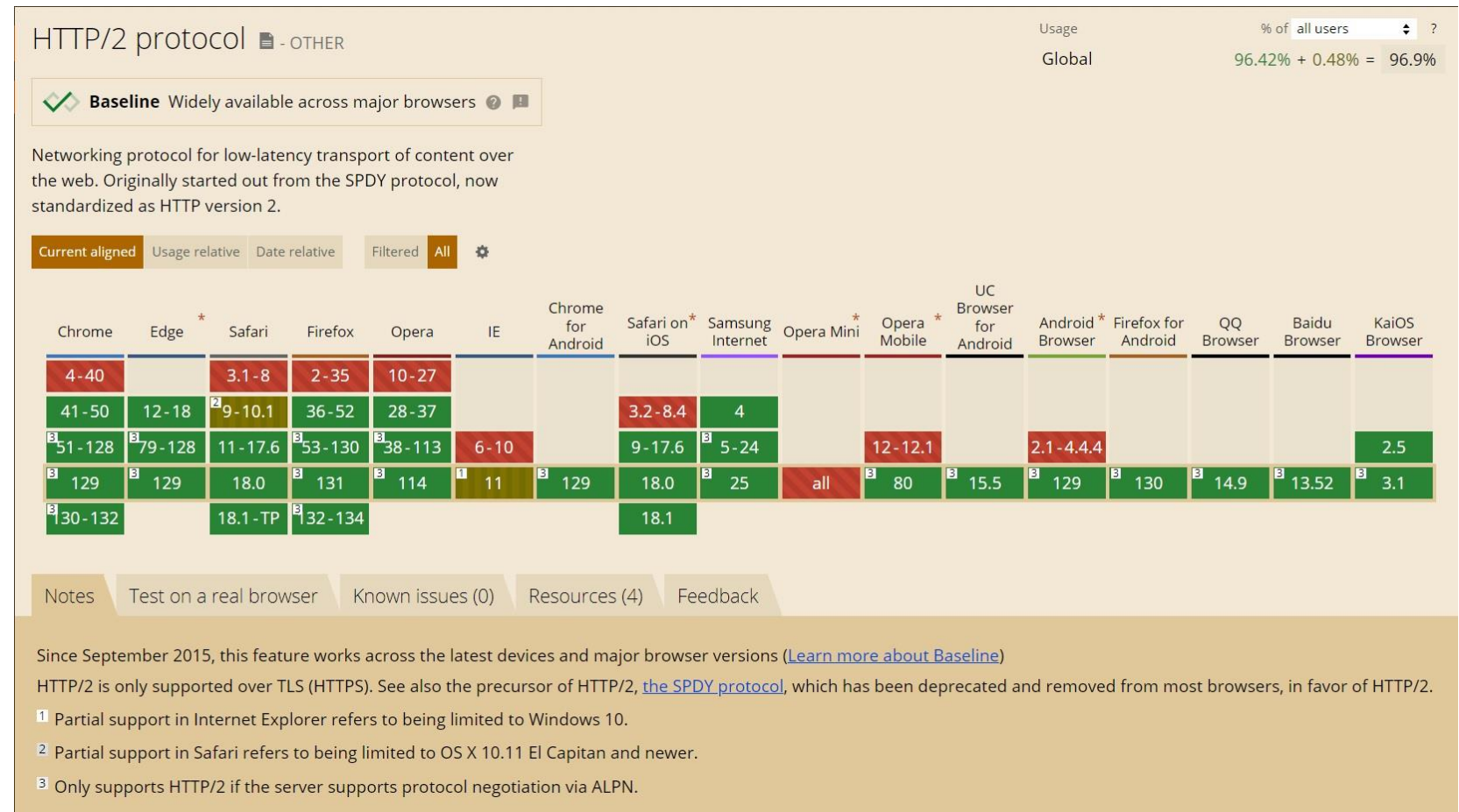
---

- ~2015: standardized
- Late 2016
  - 8.7% of all websites using it
  - 68% of all requests
- 2022:
  - > 90% of global usage
  - Includes big players: Google, YouTube, Wikipedia

# HTTP/2

## Adoption

- Rapid adoption likely because HTTP/2 requires no changes to the underlying website or application
  - Using HTTP/1.1 or HTTP/2.0 is transparent
  - Remember the Upgrade header?
- Browser and server adoption is now complete



# HTTP/2

## Security model

---

- HTTPS not mandatory
- But every implementation forces HTTPS
- HTTP/2 still different from HTTPS though!
  - You may see `https://` in your browser, but it could be using HTTP/2 (or even HTTP/3!) underneath

# Recap: Comparing HTTP

---

- HTTP/0.9
  - Non-persistent connections
  - Stateless
- HTTP/1.0
  - Non-persistent connections
  - Stateless
- HTTP/1.1
  - Persistent connections
  - Stateless
  - *Head-of-line* Blocking
- HTTP/2
  - Persistent connections
  - Stateful
  - No *Head-of-line* Blocking

# Issues

---

- *Head-of-line blocking* no longer an issue at HTTP layer, but still a problem with TCP
  - Still subject to packet loss
  - TCP will always try to reliably deliver packets in the right order
  - It will resend missing packets, which causes a delay passing data up to the layer above (HTTP)
- You know we previously said that TCP was hard to change...



# HTTP/3

---

# HTTP/3

Where to next?

---

- How do we overcome TCPs limitations?
  - Use UDP of course...
  - **HTTP3 = HTTP-over-QUIC**
- QUIC is a new transport protocol
  - QUIC streams share the same QUIC connection
    - No handshakes or slow-starts for new ones
  - QUIC streams delivered independently
    - Packet loss on one won't impact others (because of UDP)

# HTTP/3

## QUIC

---

- The use of UDP also has other benefits:
  - Internet standards are slow
  - Using UDP allows for the implementation to sit in user-space
  - Not tied to Operating System development
  - Rapidly innovate and push out versions
- HTTP streams mapped on QUIC streams with ease
  - HTTP/2 benefits without **any** head-of-line blocking
- QUIC also combines the connection handshake with TLS 1.3 handshake
  - Encryption and authentication by default

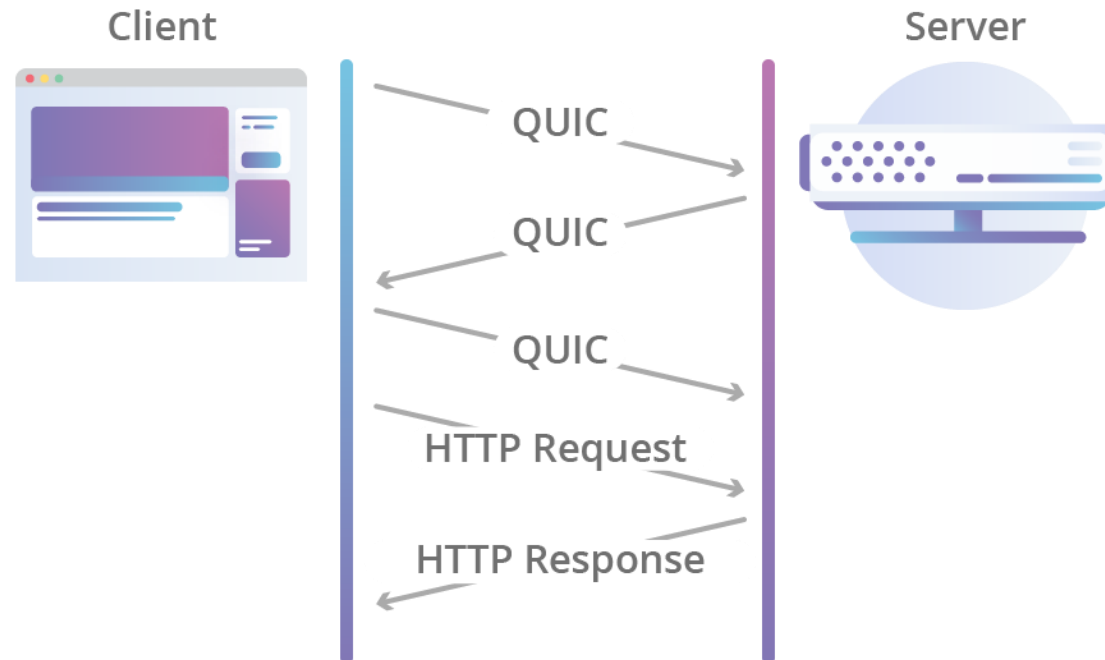


# HTTP/3

## Rapid Request

---

### HTTP Request Over QUIC



<https://blog.cloudflare.com/http3-the-past-present-and-future/>

# HTTP/3

Why not use HTTP/2 over QUIC?

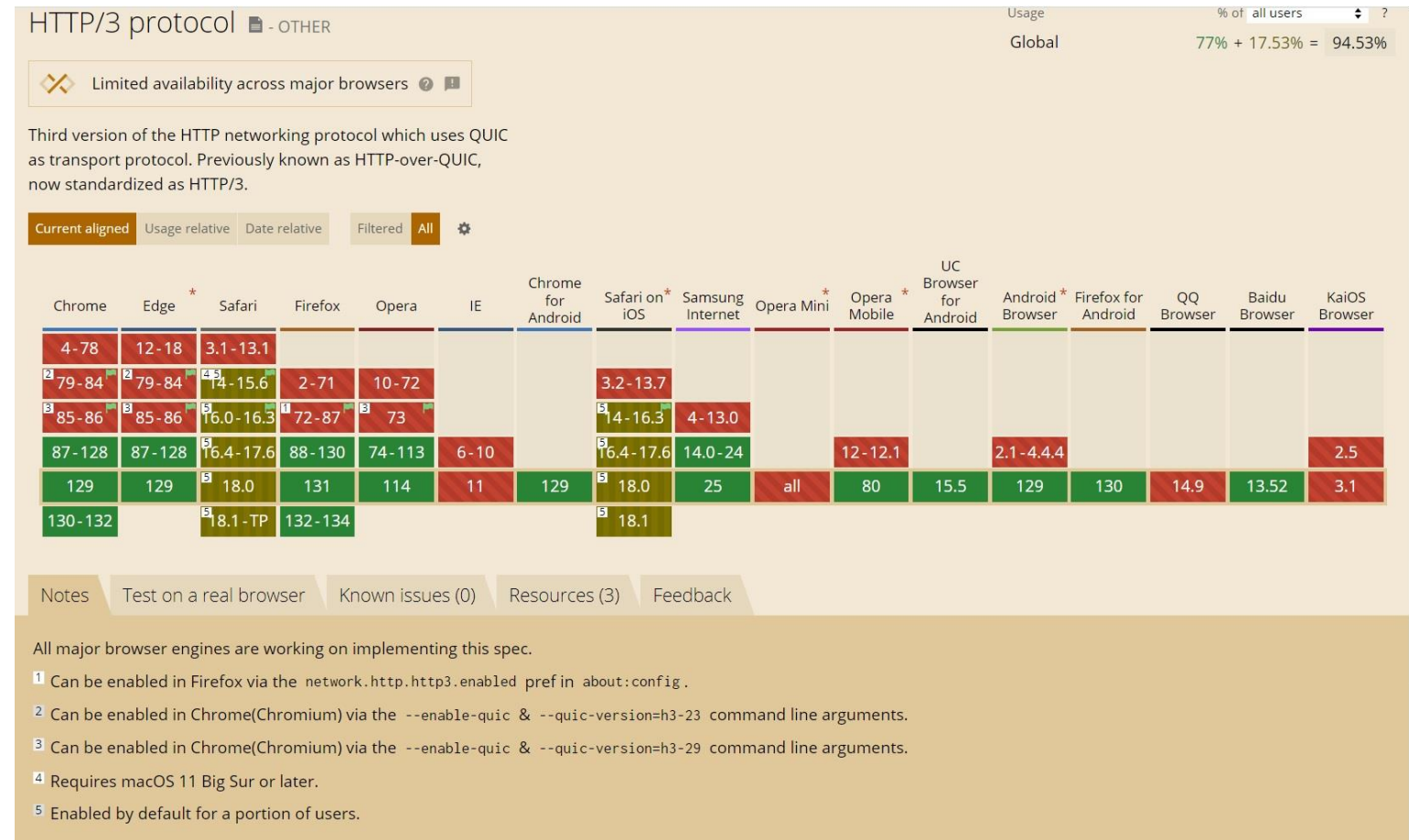
---

- Why do we need a new version at all?
- HTTP/2's header compression depends on ordering
- However, QUIC does not guarantee ordering the same way
  - A new header compression scheme was developed
  - Requires changing to the mapping
- Some functionality in HTTP/2 already offered natively with QUIC
  - Per-stream flow control
  - Dropped to reduced complexity in the protocol

# HTTP/3

## Adoption

- Browser and server adoption is still immature
  - Evolving standard



<https://caniuse.com/http3>

# HTTP/3

## Implementations

---

- <https://github.com/cloudflare/quiche>
- <https://github.com/facebook/proxygen#quic-and-http3>
- <https://github.com/mozilla/neqo>



# Recap: Comparing HTTP

---

- HTTP/0.9
  - Non-persistent connections
  - Stateless
  - TCP
- HTTP/1.0
  - Non-persistent connections
  - Stateless
  - TCP
- HTTP/1.1
  - Persistent connections
  - Stateless(ish)
  - *Head-of-line* Blocking
  - TCP
- HTTP/2
  - Persistent connections
  - Stateful
  - No *Head-of-line* Blocking (HTTP)
  - TCP
- HTTP/3
  - Persistent connections
  - Stateful
  - No *Head-of-line* Blocking
  - UDP



# Any questions?

---



# Resources

---

# Resources

---

- MDN Web Docs
  - [https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics\\_of\\_HTTP/Evolution\\_of\\_HTTP](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP)
- Cloudflare Blog
  - <https://blog.cloudflare.com/http3-the-past-present-and-future/>
- Chrome Web Dev
  - <https://web.dev/performance-http2/>