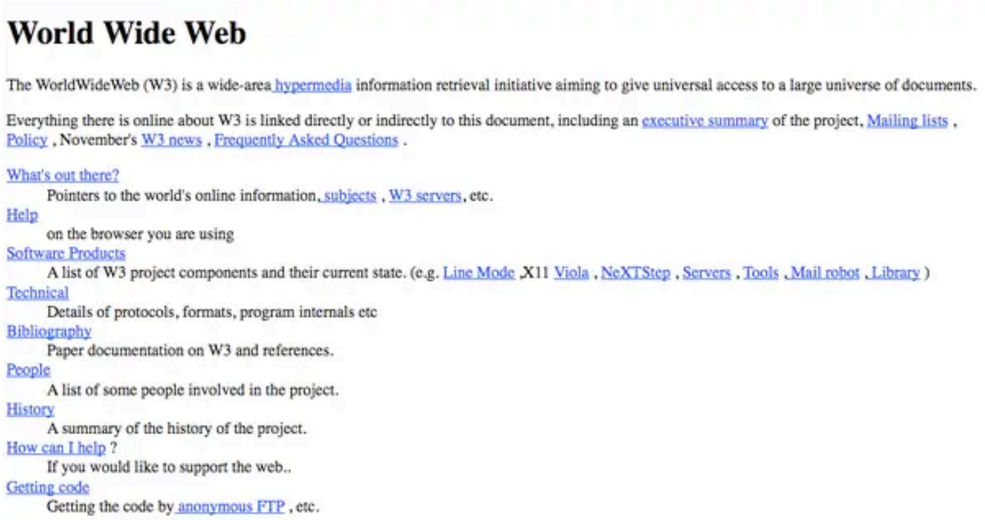
Modern Web Protocols

cs249i

Evolution of the Web

- Earliest websites provided static content with little additional media



World Wide Web

The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an executive summary of the project, Mailing lists , Policy , November's W3 news , Frequently Asked Questions .

What's out there?
Pointers to the world's online information, subjects , W3 servers, etc.
Help
on the browser you are using
Software Products
A list of W3 project components and their current state. (e.g. Line Mode ,X11 Viola , NeXTStep , Servers , Tools , Mail robot , Library)
Technical
Details of protocols, formats, program internals etc
Bibliography
Paper documentation on W3 and references.
People
A list of some people involved in the project.
History
A summary of the history of the project.
How can I help ?
If you would like to support the web..
Getting code
Getting the code by anonymous FTP , etc.

Ostensibly the first website ever

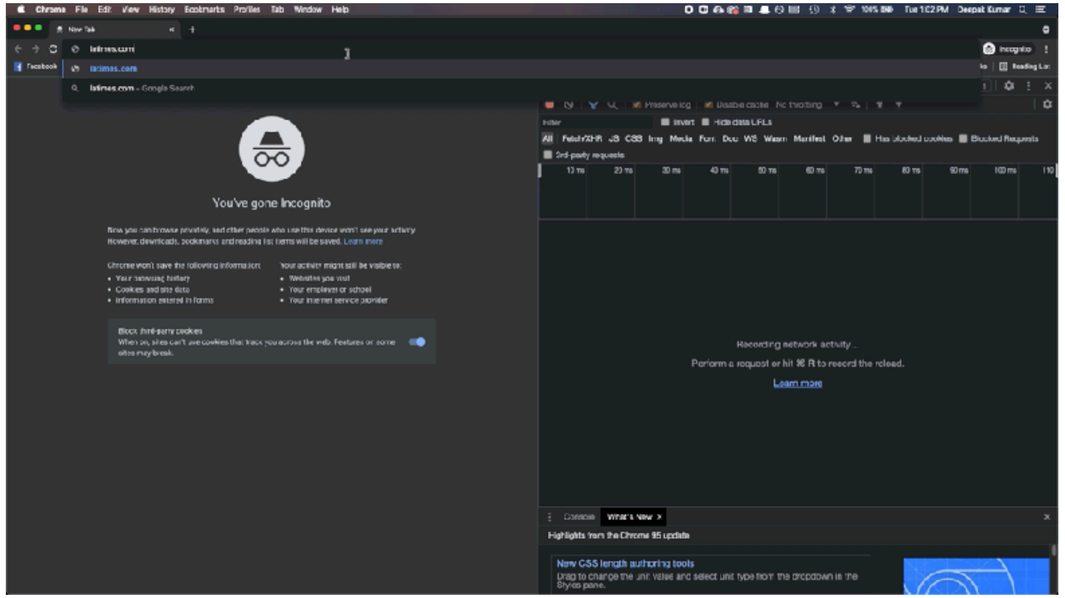
Evolution of the Web

- Earliest websites provided static content with little additional media
- Over time, websites grew to include many more things, like deepening the web structure (adding more pages), adding images, logos, and even started serving some *dynamic content*

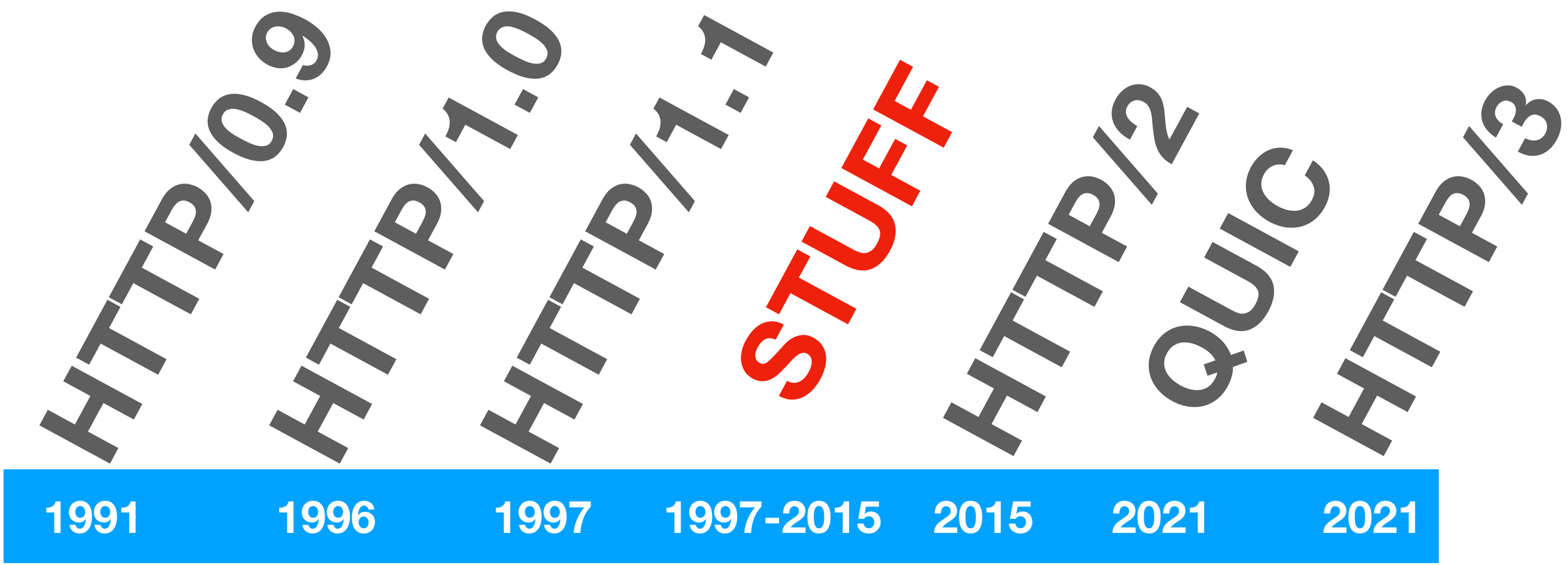
ESPN in 1996

Evolution of the Web

- Earliest websites provided static content with little additional media
- Over time, websites grew to include many more things, like deepening the web structure (adding more pages), adding images, logos, and even started serving some *dynamic content*
- Modern websites are incredibly complex are rely on often hundreds of resources to properly function



A History of Web Protocols



Websites are growing up. So should our protocols.

Interfacing with the Web

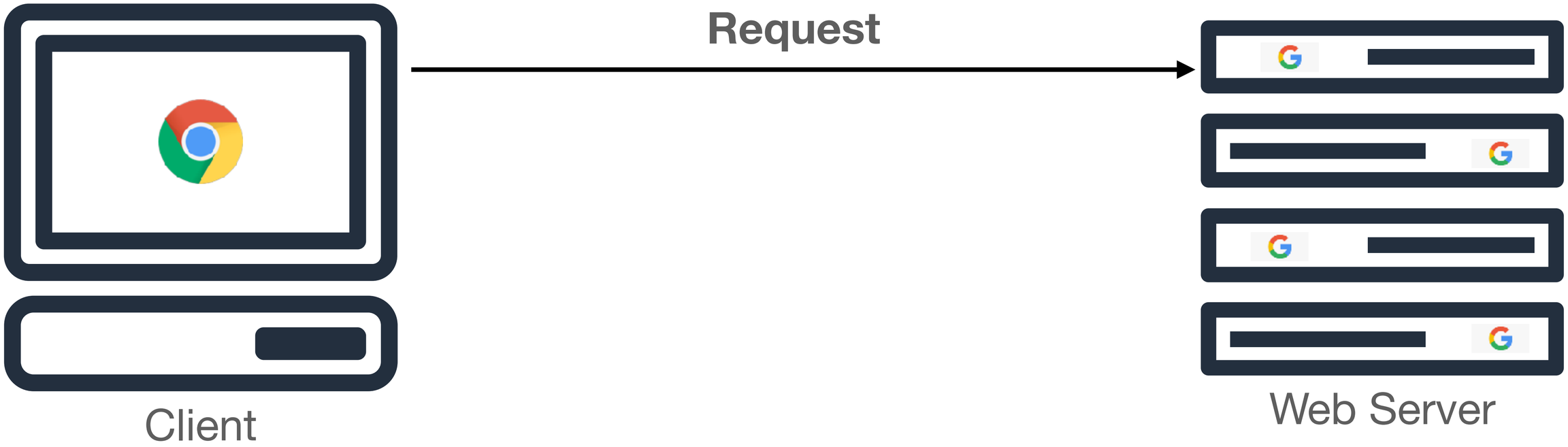
Client / Server Model

Client

Web Server

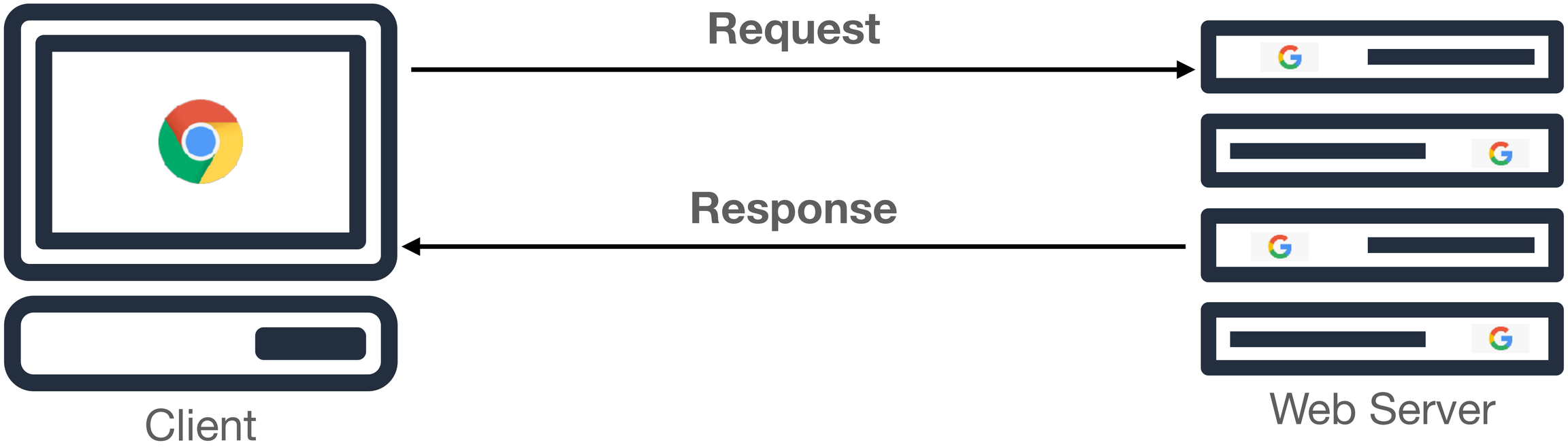
Interfacing with the Web

Client / Server Model

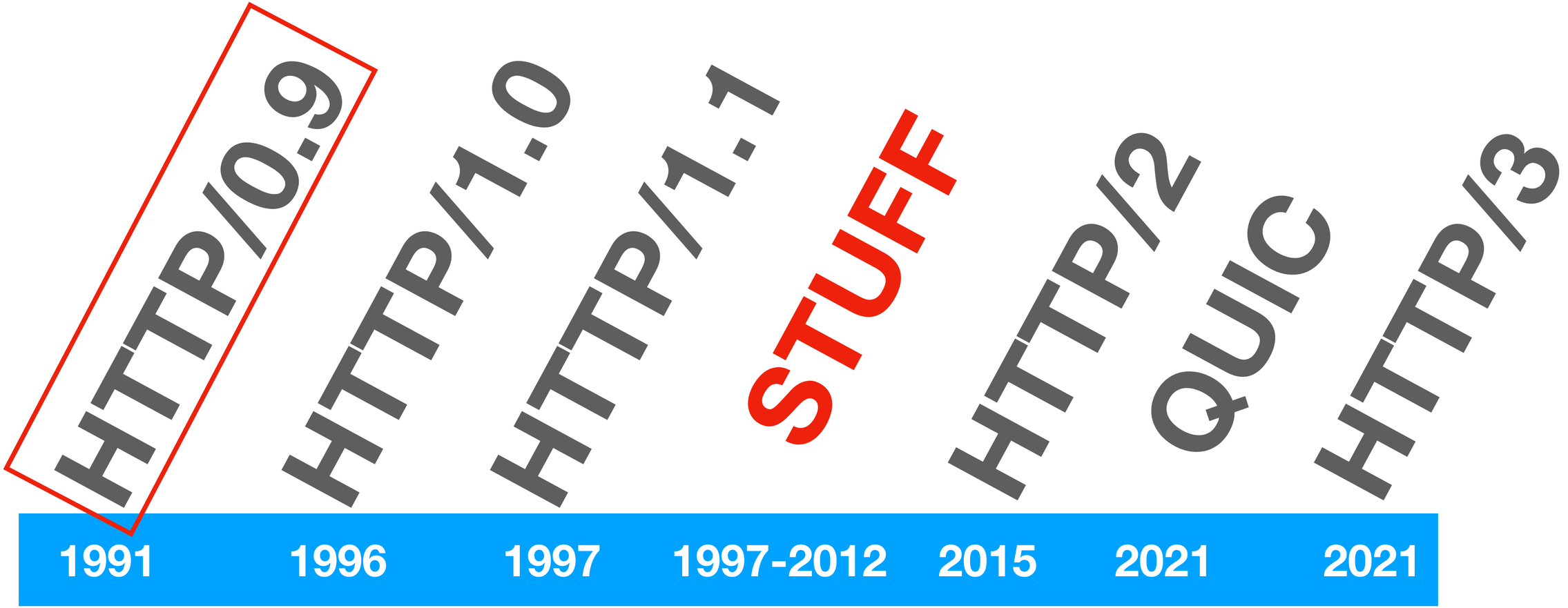


Interfacing with the Web

Client / Server Model



A History of Web Protocols



HTTP/0.9

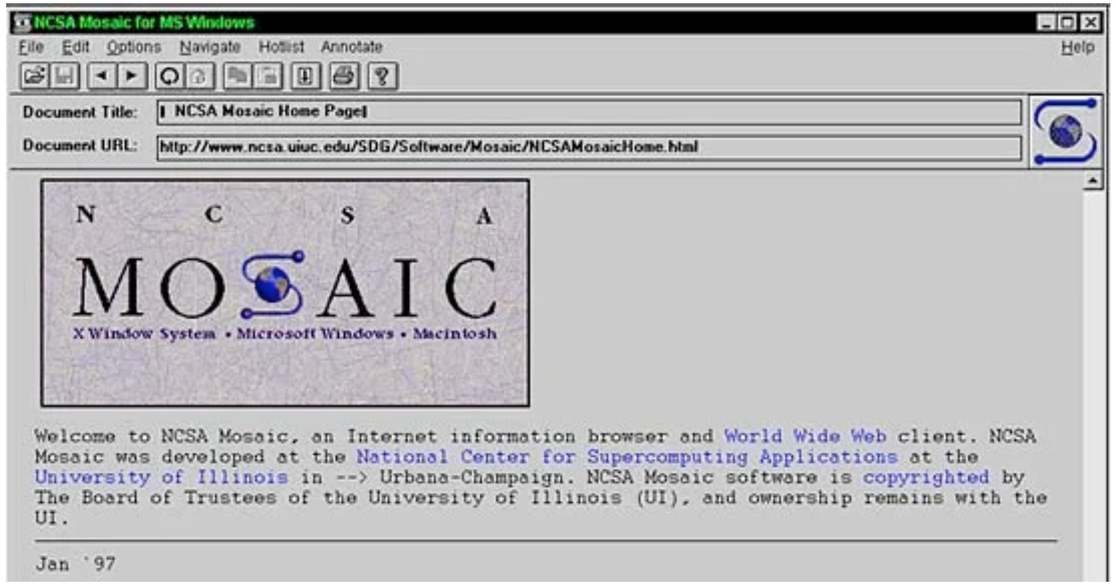
Single Line Protocol

- In 1991, Tim-Berners Lee needed a simple protocol to test his new invention (the web)
- Request was a single line command, supported *only* retrieving HTML content
  - **GET /index.html**
- Response was the file data itself!
- HTTP/0.9 was built on top of **TCP**, for reliable transport of data, and the connection was closed after every single request

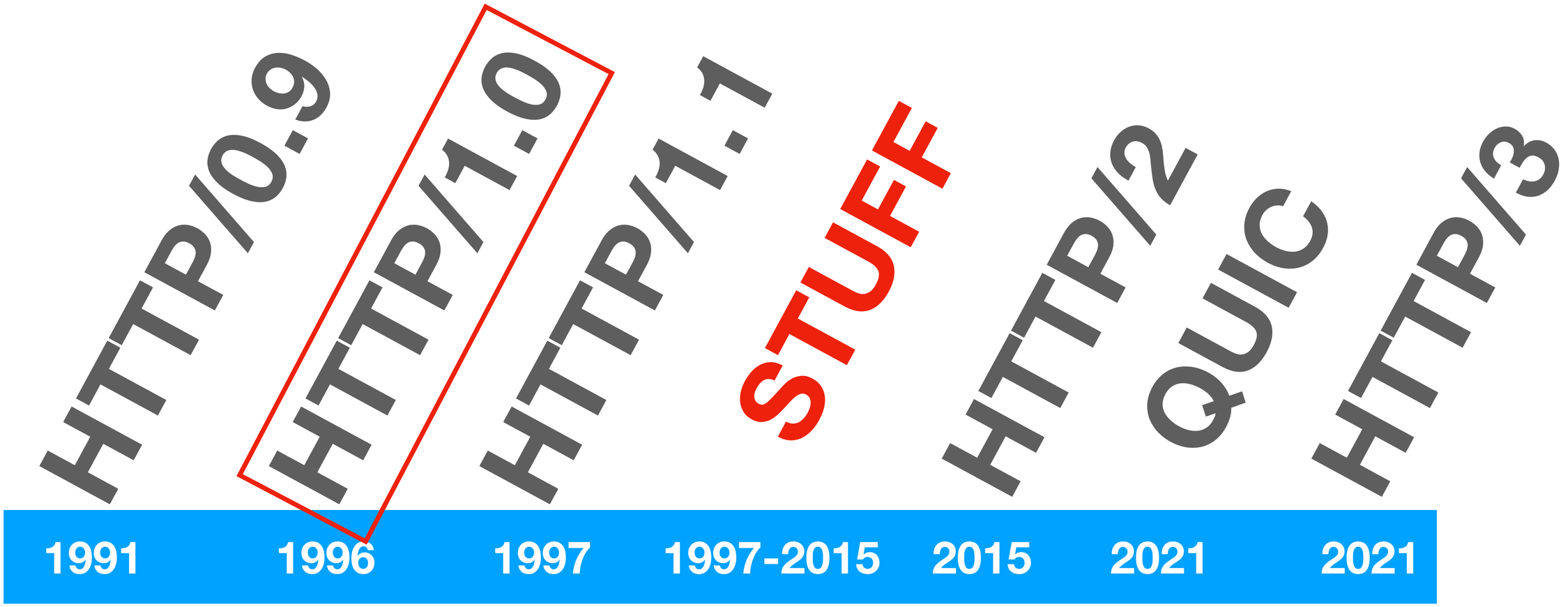
The Web Catches On

Moar Content

- The web started catching on, and people started to build out software that could interact with other types of content (e.g., images) and share other types of meta-data
- HTML specification started to show a lot of progress
- The first browsers started showing up around 1994 – Netscape (first browser) was developed as an academic project at NSCA in Champaign, IL
  - Began the first “browser wars”



A History of Web Protocols



HTTP/1.0

Specification Improvements

- Goals: “generic, **stateless**, object-oriented protocol which can be used for many tasks, such as *name servers* and *distributed object management systems*” (from RFC1945)

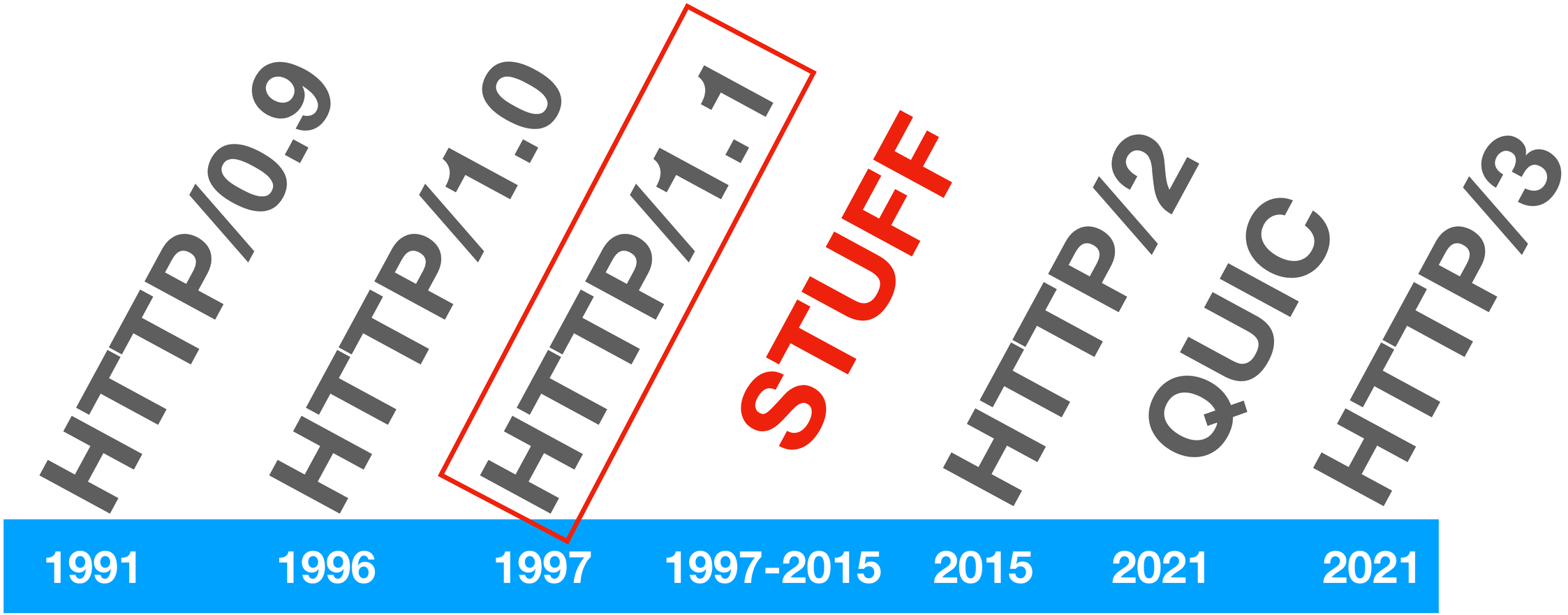
- Added versioning, a number of new methods (POST, HEAD, PUT, DELETE, LINK, UNLINK), supported myriad different content-types (no longer just HTML!), and included *headers* to accompany each request and response

HTTP/1.0

Mired with Problems

- Connections were closed after requesting a single resource, this made things frustrating, slow, and expensive as websites started hosting dozens of files
    - Internet connection speeds were slow, and TCP slow start had just been rolled out widely
- People wanted to host multiple websites at the same IP address, which wasn’t possible (e.g., colocation services)

A History of Web Protocols



HTTP/1.1

A New Era

- HTTP/1.1 fixed many problems and challenges with early versions of the protocol
  - Added the Host header (to enable multiple websites with different domains to be served from the same IP address)
  - Allowed for **persistent connections**
    - Allowed chunked responses
    - Enabled pipelining of requests

HTTP/1.1

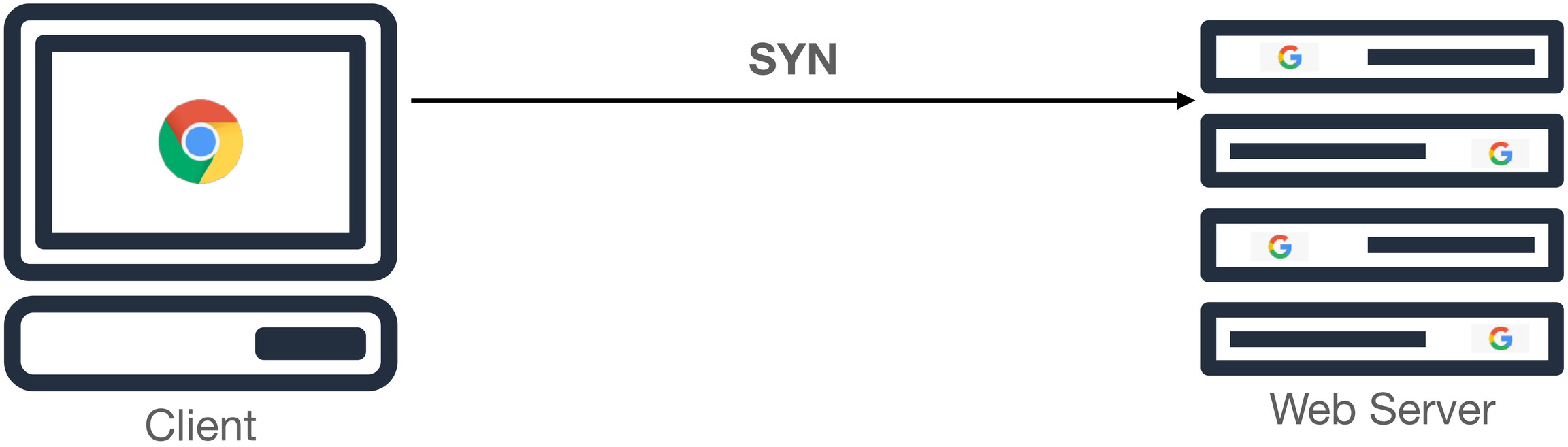
Persistent Connections

Client

Web Server

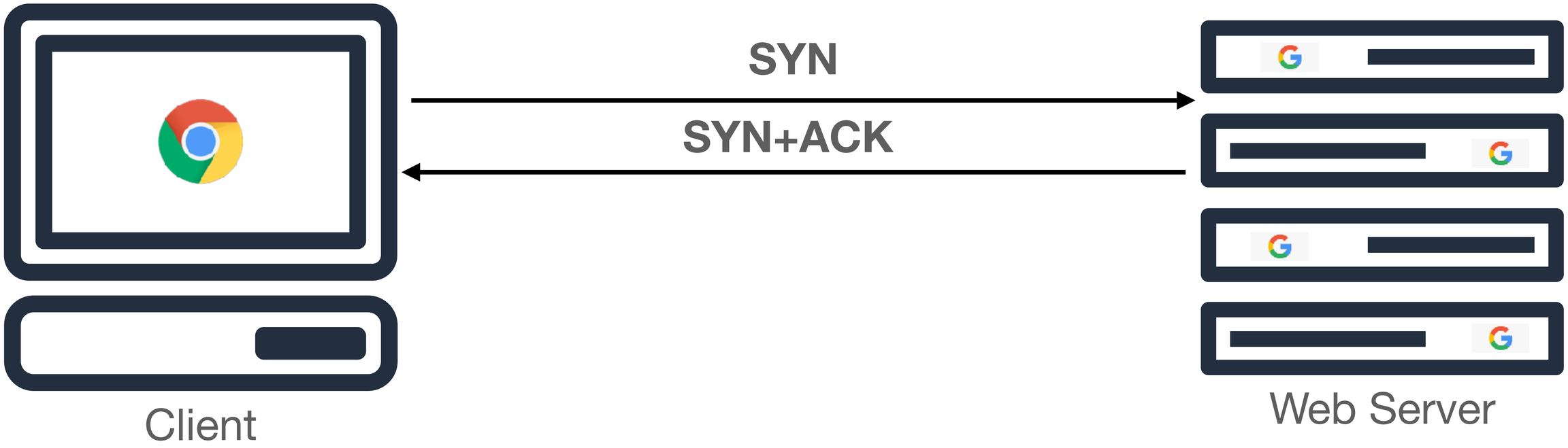
HTTP/1.1

Persistent Connections



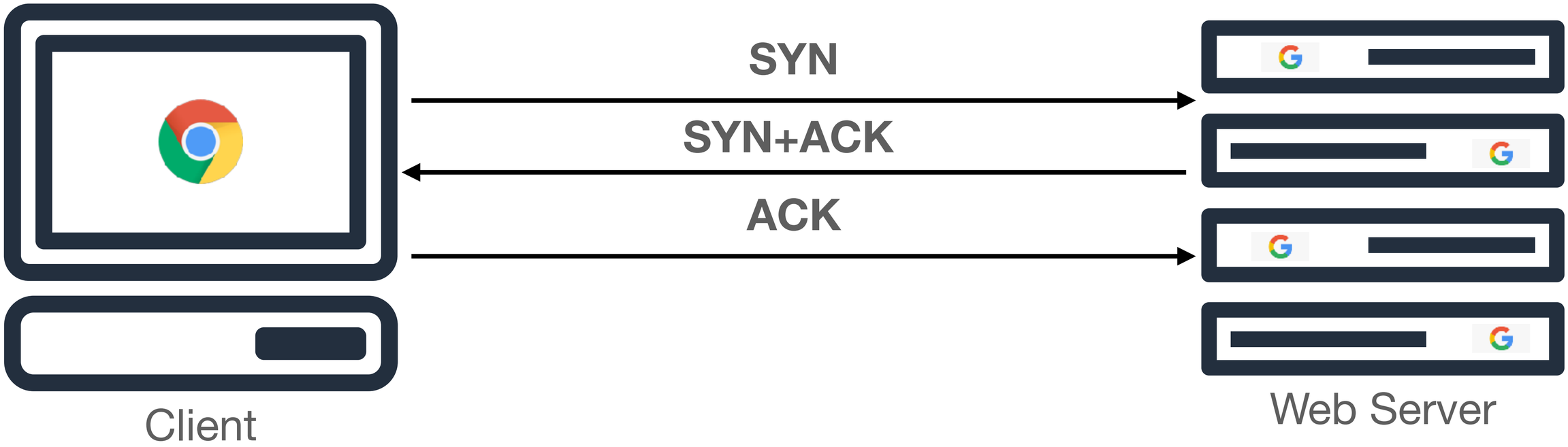
HTTP/1.1

Persistent Connections



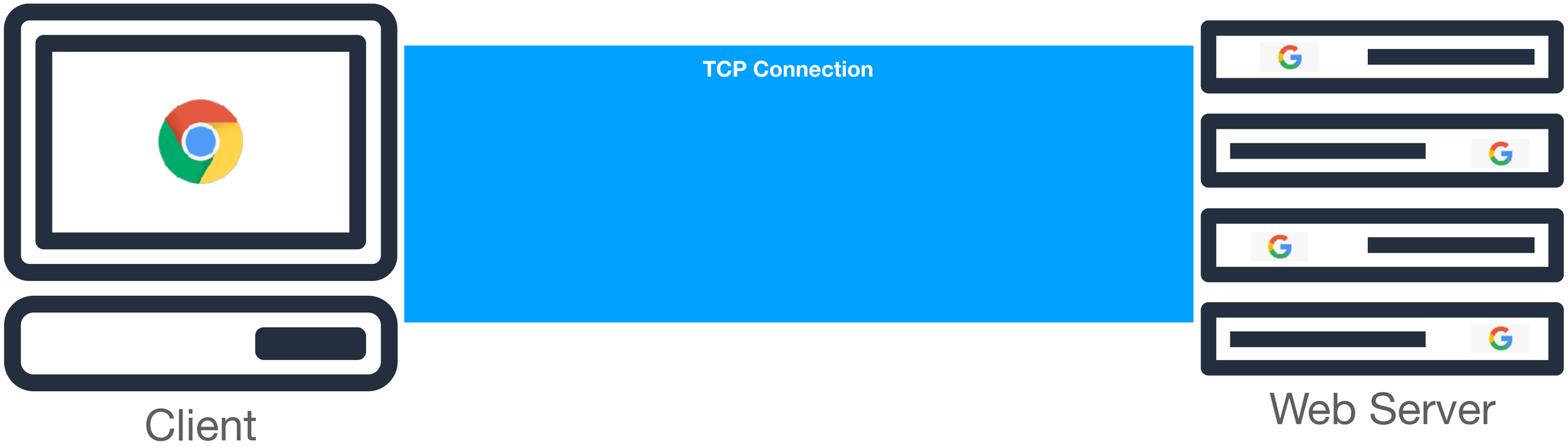
HTTP/1.1

Persistent Connections



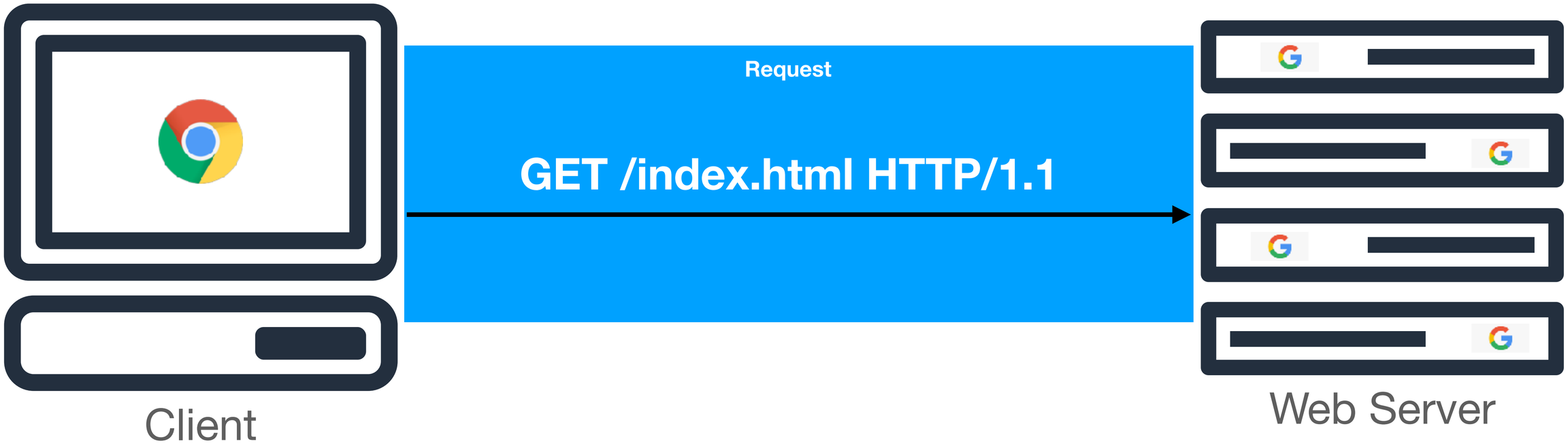
HTTP/1.1

Persistent Connections



HTTP/1.1

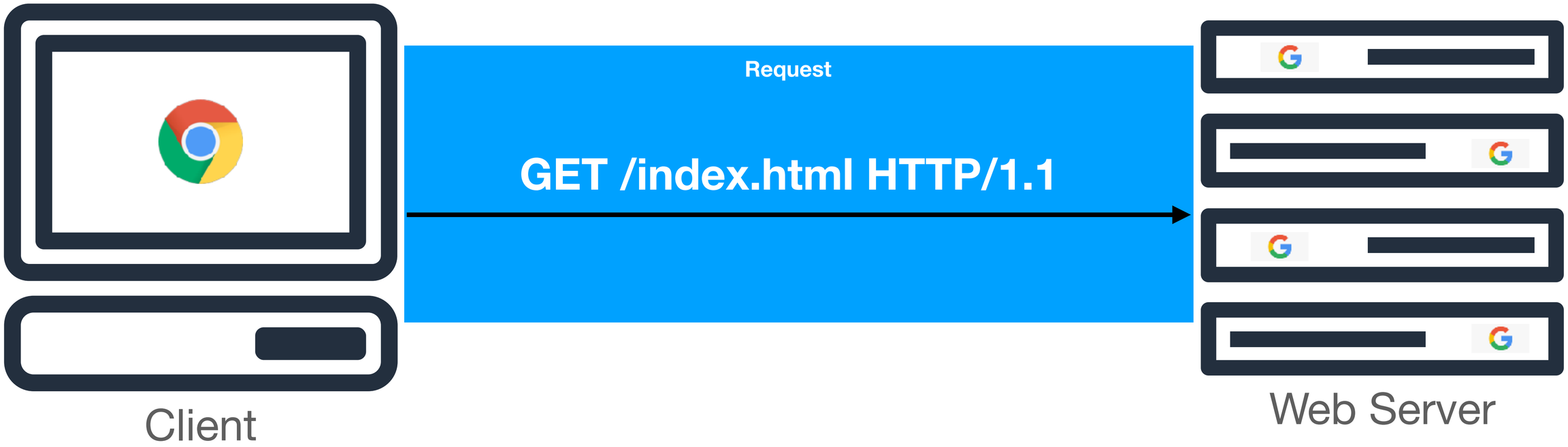
Persistent Connections



HTTP/1.1

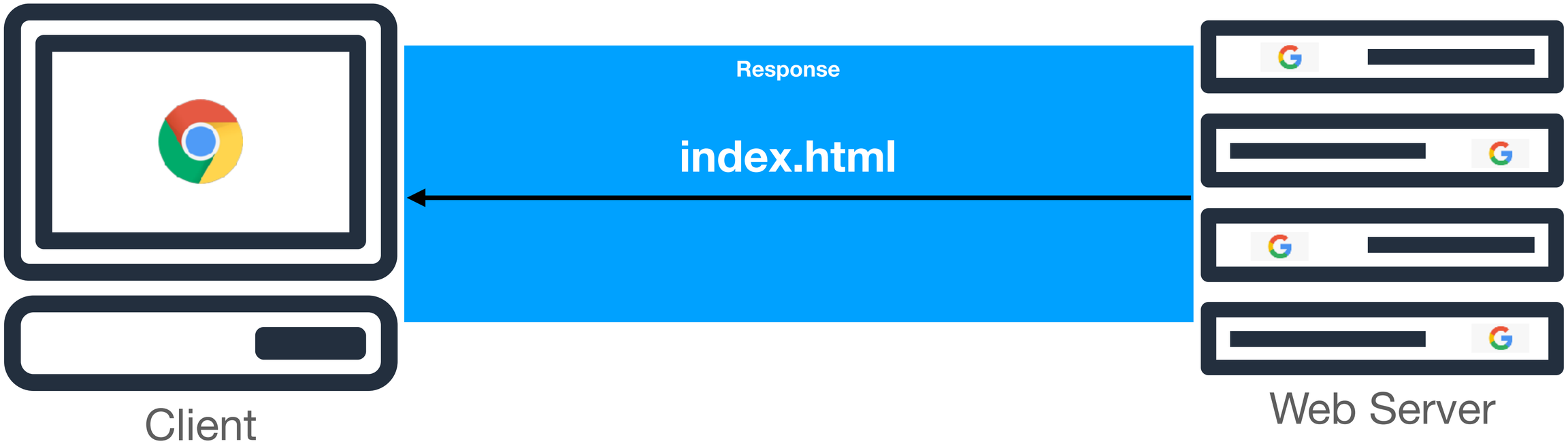
Persistent Connections

```
GET /index.html HTTP/1.1
Host: kumarde.com
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://developer.mozilla.org/testpage.html
Connection: keep-alive
```



HTTP/1.1

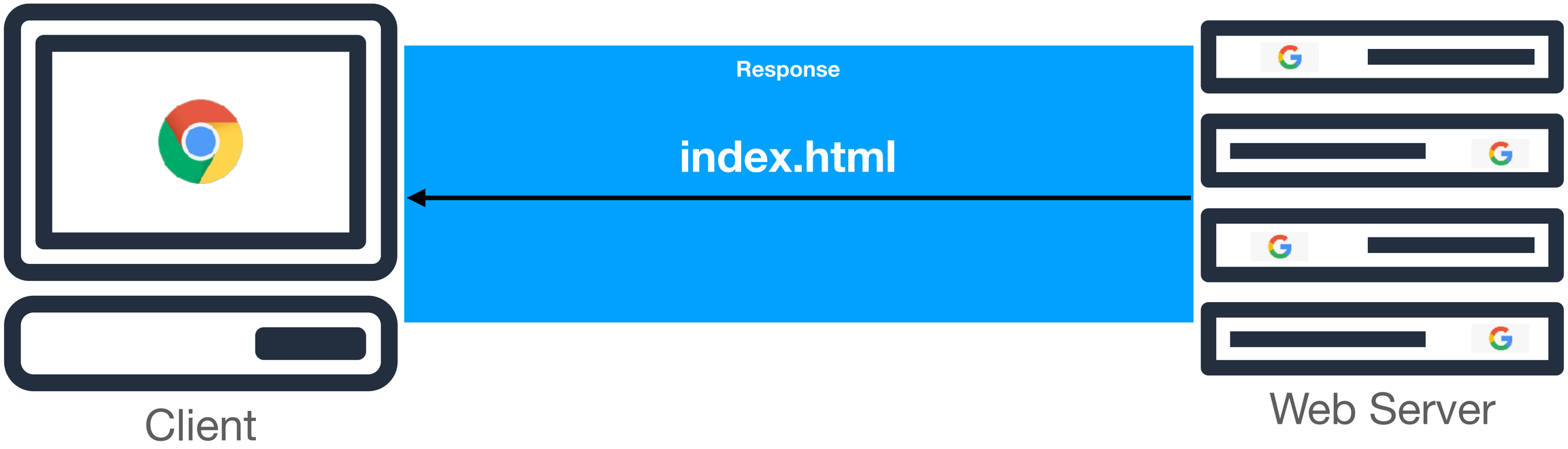
Persistent Connections



HTTP/1.1

Persistent Connections

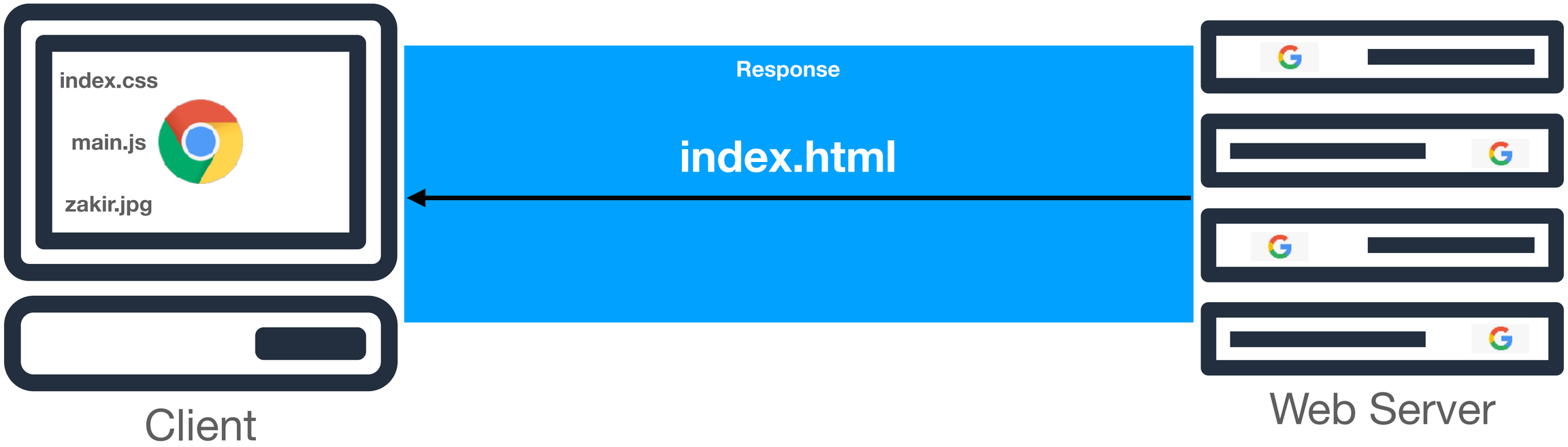
```
200 OK
Access-Control-Allow-Origin: *
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Date: Mon, 18 Jul 2016 16:06:00 GMT
Keep-Alive: timeout=5, max=997
Last-Modified: Mon, 18 Jul 2016 02:36:04 GMT
Server: Apache
Transfer-Encoding: chunked
```



HTTP/1.1

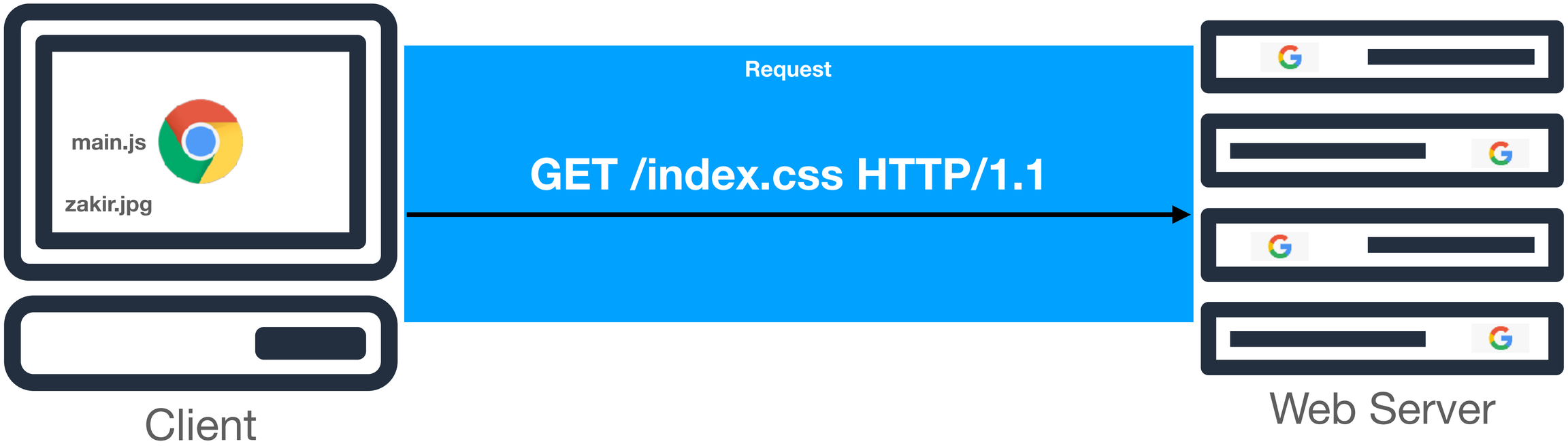
Persistent Connections

```
200 OK
Access-Control-Allow-Origin: *
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Date: Mon, 18 Jul 2016 16:06:00 GMT
Keep-Alive: timeout=5, max=997
Last-Modified: Mon, 18 Jul 2016 02:36:04 GMT
Server: Apache
Transfer-Encoding: chunked
```



HTTP/1.1

Persistent Connections



HTTP/1.1

Persistent Connections

HTTP/1.1

Persistent Connections

Modern browsers will open up to 6 TCP connections per host, plus 4 external TCP connections at a time

HTTP/1.1

Chunking

- With persistent connections, servers could also now **chunk** data by sending a `Transfer-Encoding: Chunked` header
- Essentially, this means that servers can break up their responses into independent chunks – each chunk does not need to know about the other chunks in order to send correctly (e.g., this is good for TCP)
    - This enabled the transfer of large files via HTTP, and also enabled streaming data (e.g., video content streaming, which is typically TCP based)

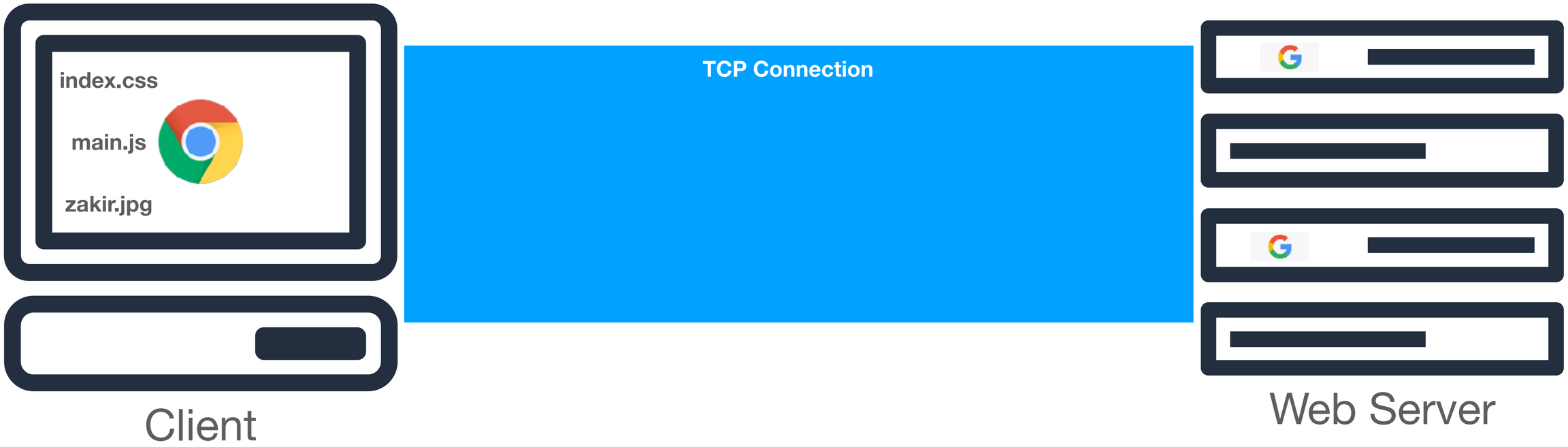
HTTP/1.1

Pipelining

- Another great feature for HTTP innovation was *pipelining*, essentially the ability for clients to make additional requests before the response to previous requests arrived
- Requirement: Servers needed to send back responses *in the order* they were received
  - HTTP/1.1 specification dictated that servers **MUST** implement pipelining
- On the server side, this simply amount to keeping network buffers open and know to look for more HTTP requests on the TCP connection before response
- Clients did **not** want to deal with HTTP pipelining… why?

HTTP/1.1

Pipelining



HTTP/1.1

Pipelining

Without pipelining, I have to open three separate TCP connections

HTTP/1.1

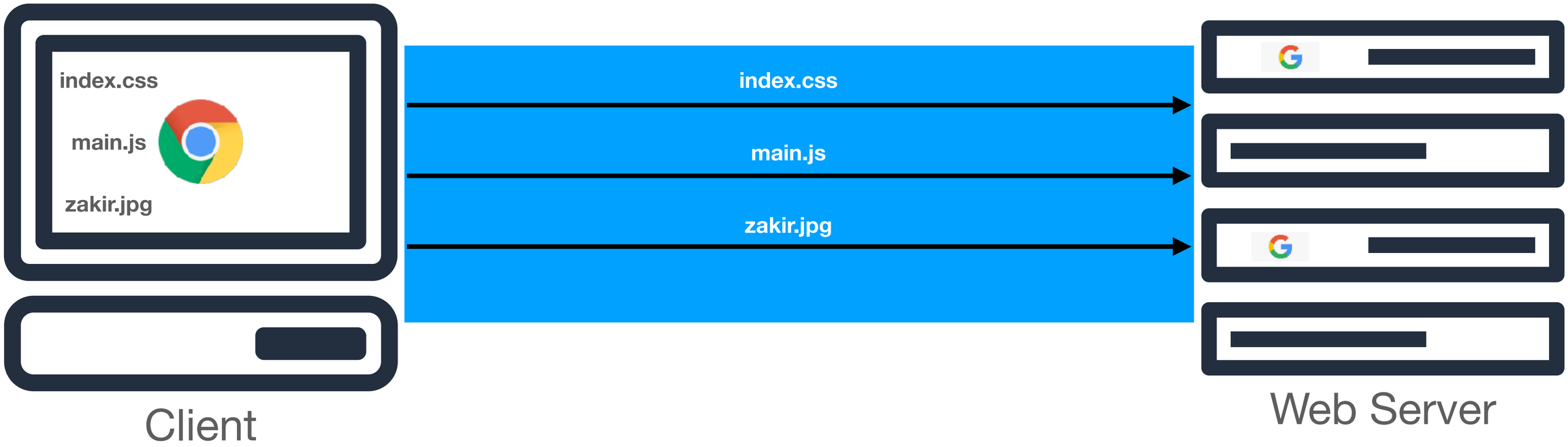
Pipelining

With pipelining, I can use just one TCP connection!

HTTP/1.1

Pipelining

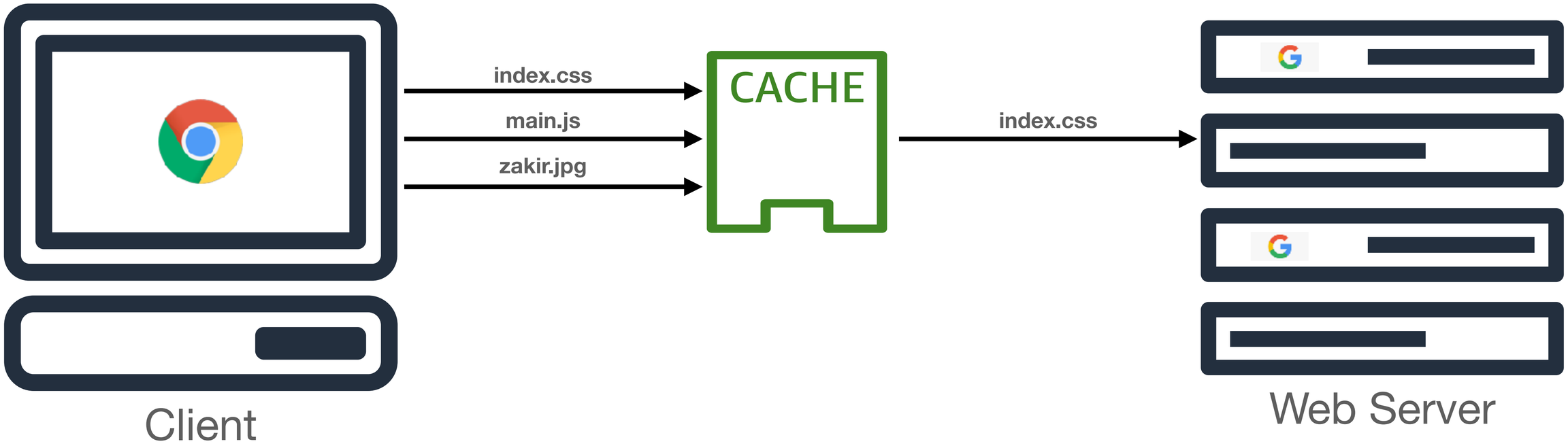
But what happens if index.css takes a long time to retrieve?



HTTP/1.1

Pipelining

Also, what about HTTP proxies?



HTTP/1.1

Head of Line Blocking

- Big problem with HTTP/1.1 pipelining is a concept called head of line blocking (HOL) which essentially means that subsequent resources on a shared connection need to **wait for the first request to be serviced** before they can be served
- In theory, pipelining is a good idea, but there are some thorny edge cases
  - If proxies do not support pipelining, clients need to retransmit or fall-back to non-pipelining, which is hard to identify and causes delays
  - This crippled HTTP/1.1 pipelining, so much so that no browsers currently support it and browser developers get angry when you bring it up

HTTP/1.1

Head of Line Blocking

> Pipelining has been an undeniable pain in the ass. Nobody has gotten it working properly without hacks and even then problems pop up. It should work, but it is nonetheless a mess. It would be nice if we could get it running now, but obviously that hasn't happened. THERE IS NO POINT IN DEBATING THIS. Yes, servers should be fixed, but they aren't. Yes, heuristics to get it working are possible, but they're still not idiot-proof. There is nothing productive in rambling on this topic here.

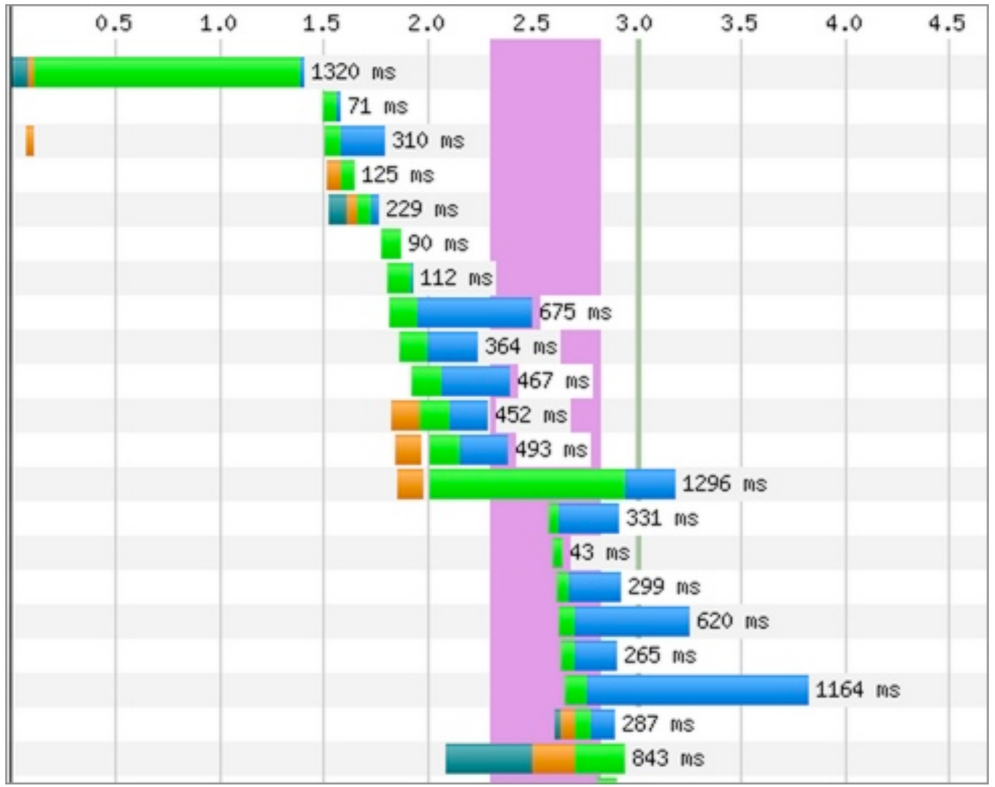
https://bugzilla.mozilla.org/show_bug.cgi?id=264354

HTTP/1.1

Head of Line Blocking

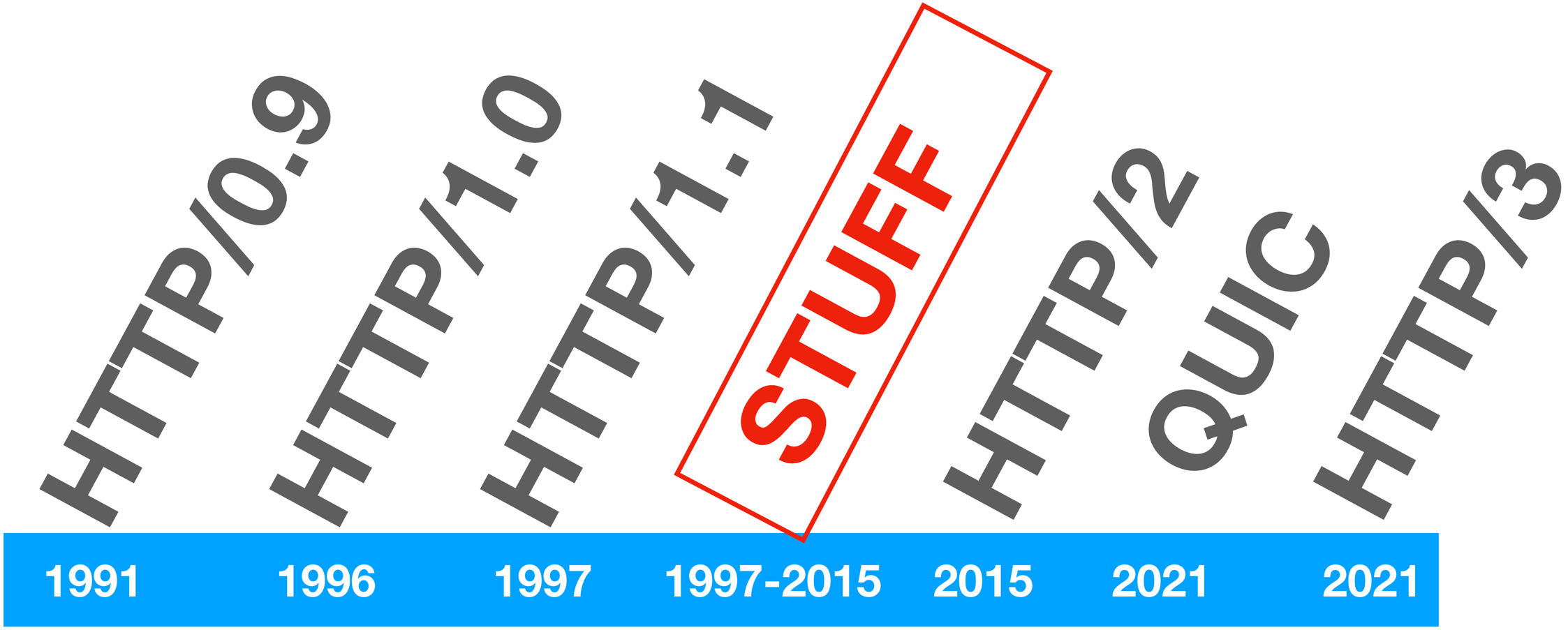
- Head of Line blocking is **broader** than pipelining – today, modern browsers still only open a maximum of 6 connections and have to wait for requests to finish before issuing new ones

- This is obviously slow,



https://www.merkleinc.com/blog/http2-electric-boogaloo-0

A History of Web Protocols



Stuff that Happened from 1997 – 2012

Context

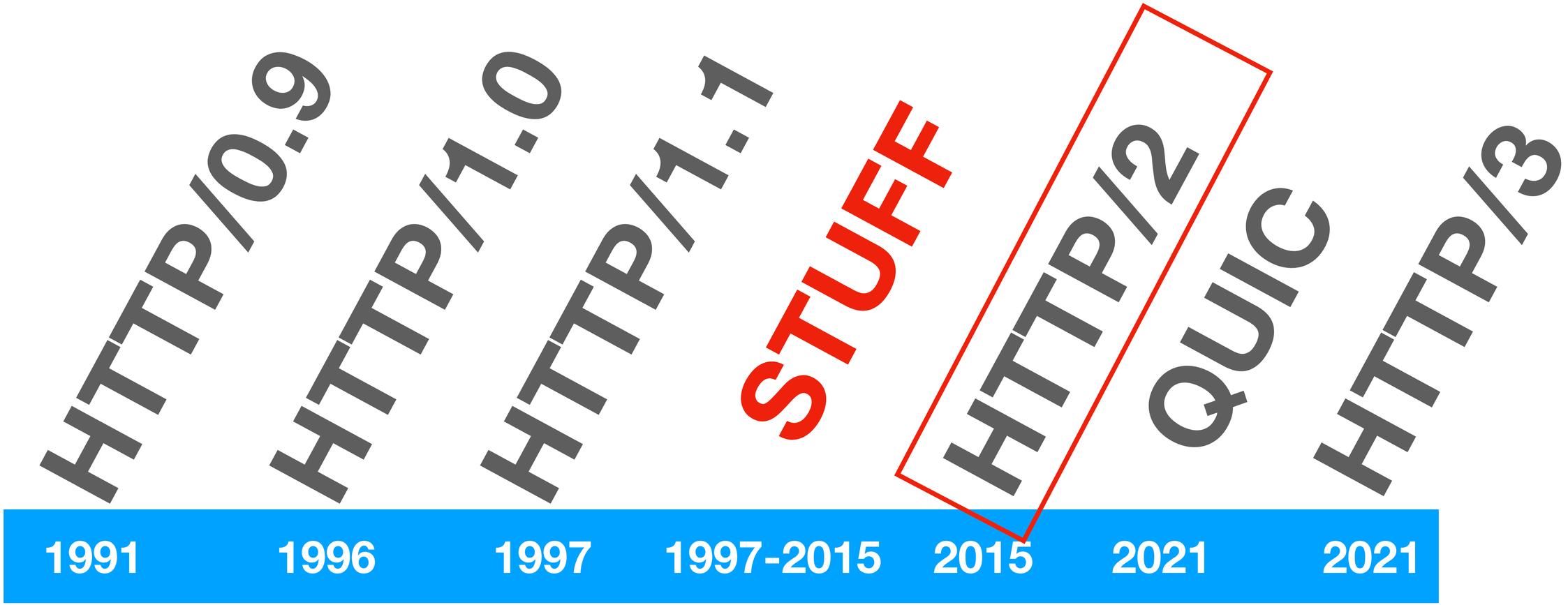
- Modern websites **exploded** with dynamic content and an increased reliance on web resources to provide new online experiences
    - In 2011, median number of requests per modern webpage was 40, with some requesting up to 100 different
- Internet speeds and infrastructure significantly improved, networks matured quite a lot
    - **Millions** of people were accessing the Internet (and the web) for the first time, adding significantly to load
- We needed to figure out how to meet the demands of a **growing web, and HTTP/1.1 was not cutting it.**

SPDY

Google's solution

- Google engineers decided to try and modernize how web content was shared, and developed SPDY (pronounced "speedy"), which was largely motivated by reducing page load times for websites
- SPDY was a *translation layer* between HTTP clients and servers and sat in front of HTTP on both ends
  - Shipped in Chrome, Firefox also implemented SPDY shortly after
- At its peak, SPDY served the majority of traffic to Google services and a whole host of other Internet services
- SPDY formed the foundation for what would eventually be HTTP/2, SPDY is now deprecated

A History of Web Protocols



HTTP/2

Design Goals

1. Eliminate HoL blocking by multiplexing HTTP requests over a single TCP connection

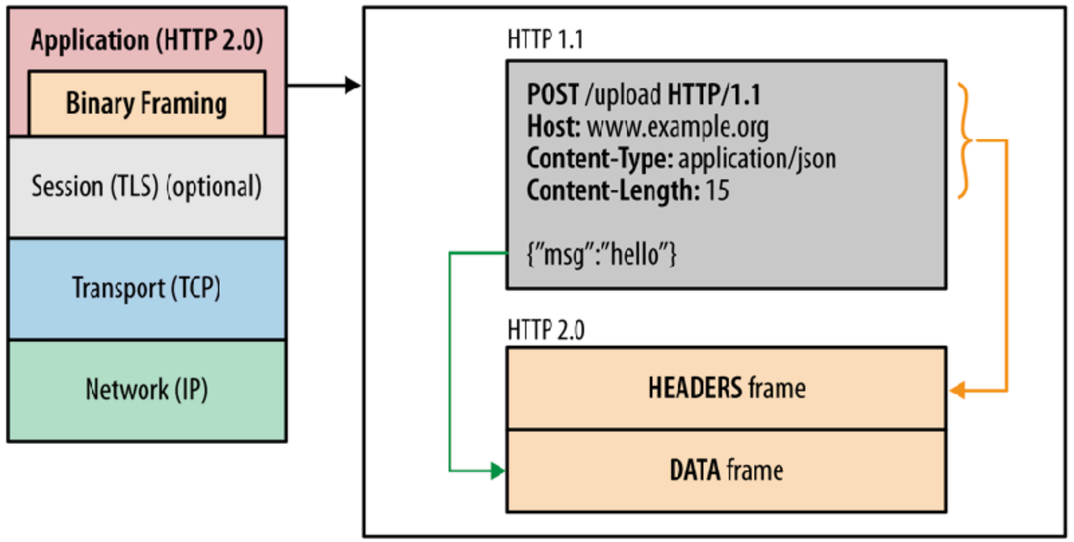
2. Give servers more agency (e.g., allow them to *push* content over persistent connections)

3. Reduce unnecessary duplicate bytes sent over the wire (e.g., static headers)

HTTP/2

Goal 1: Multiplexing Requests

- Core idea: Move *away* from an ASCII-based request / response cycle for data transfer, and move towards a *binary stream of data*
  - **Not** backwards compatible with HTTP/1.x
- New terminology
  - Streams: A bidirectional flow of bytes which can carry one or more messages, denoted by an integer ***stream_id***
  - Message: Complete sequence of frames that map to a logical request or response
  - Frame: Smallest unit of data, can contain either header information or content information

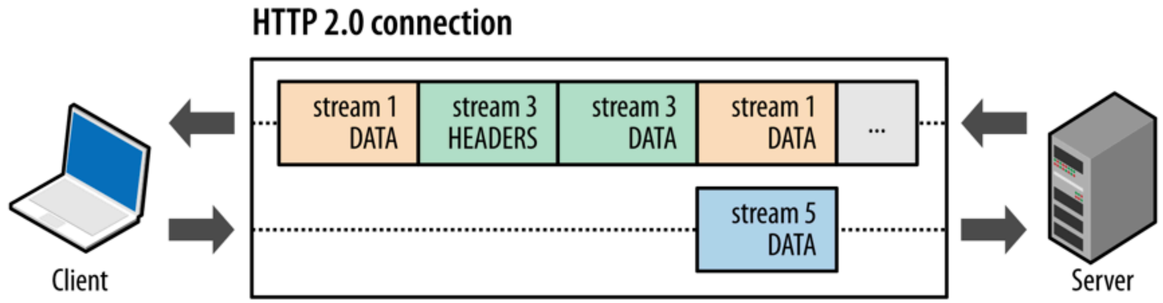


https://developers.google.com/web/fundamentals/performance/http2

HTTP/2

Goal 1: Multiplexing Requests

- HTTP/2 uses a **single** TCP connection for any number of arbitrary HTTP requests and responses
  - Everything is logically separated by stream_id (4 byte integer)
- This means that if the server takes significant amounts of time for one request (say, the first one), other requests can still be completed while we wait for that one!



https://developers.google.com/web/fundamentals/performance/http2

HTTP/2

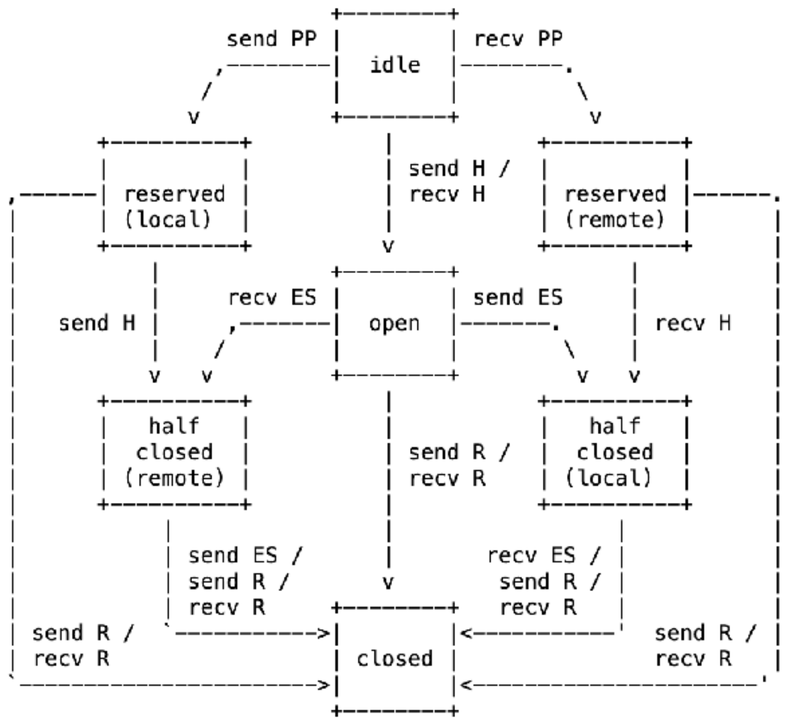
Goal 1: Multiplexing Requests

HTTP/2

Stream Lifecycle

- All streams start in an “idle” state
- Upon receiving or sending a HEADERS Frame (H), the stream becomes open, and anyone can send or receive data on it
  - Can send frames of any type on the stream until an END_STREAM (ES) flag is sent
  - RFC specifies that one full HTTP request / response pair must put the stream into a “closed” state

The lifecycle of a stream is shown in Figure 2.



send: endpoint sends this frame
recv: endpoint receives this frame

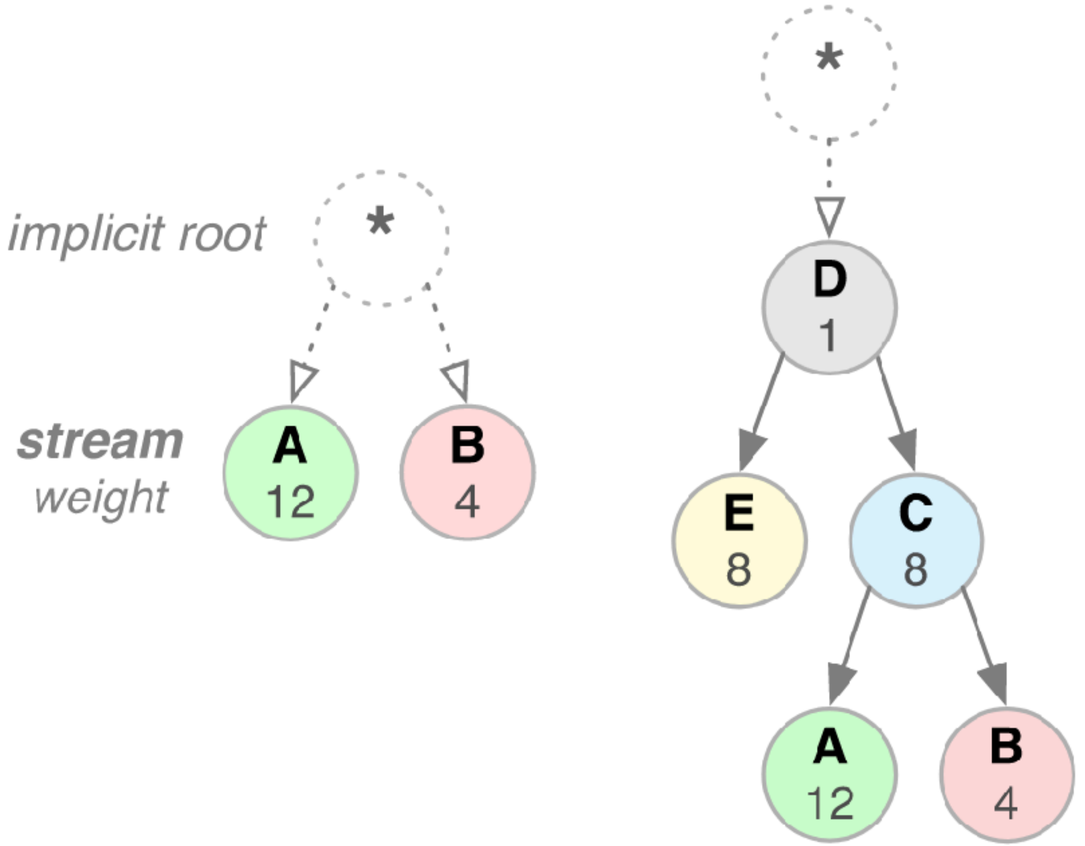
H: HEADERS frame (with implied CONTINUATIONs)
PP: PUSH_PROMISE frame (with implied CONTINUATIONs)
ES: END_STREAM flag
R: RST_STREAM frame

https://datatracker.ietf.org/doc/html/rfc7540

HTTP/2

Stream Prioritization

- Either the client or server can create a new stream, but the ordering of streams may matter to some applications
- HTTP/2 also support *prioritization of streams,* which is a mechanism that allows the client to ask for specific streams ahead of the streams
  - Clients can build a stream prioritization tree, which is essentially weights on a graph sent to the server along with each stream request
- Asking the server: “If you can, please process stream 8 before you process stream 12”, but it’s not a guarantee



HTTP/2

Design Goals

✅ **Eliminate HoL blocking by multiplexing HTTP requests over a single TCP connection**

2. Give servers more agency (e.g., allow them to *push* content over persistent connections)

3. Reduce unnecessary duplicate bytes sent over the wire (e.g., static headers)

HTTP/2

Goal 2: Giving servers more agency

- HTTP/2 offers a new feature called Server Push, which enables the server to send data to the client that *it hasn't even requested yet*.
  - **Why might we want this?**

HTTP/2

Goal 2: Giving servers more agency

- HTTP/2 offers a new feature called Server Push, which enables the server to send data to the client that *it hasn't even requested yet*.
  - **Why might we want this?**
- Despite the fact that websites are highly dynamic, they still serve lots of static content
  - e.g., index.css, main.js
- The server knows the client will need these assets to load the page, so why not just give it to them in advance?

HTTP/2

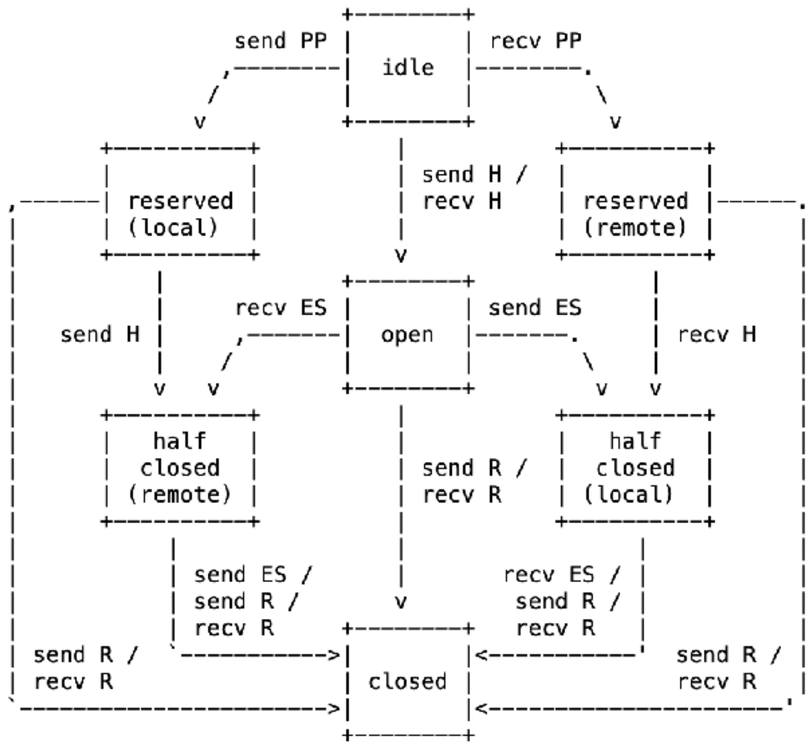
Goal 2: Giving servers more agency

HTTP/2

Goal 2: Giving servers more agency

- Server Push is implemented using a PUSH_PROMISE frame on a new stream
  - Essentially asking to reserve an HTTP/2 stream for pushing additional data to the client
- Clients can still, however, reject the push by sending a RST_STREAM frame, which means “I don’t want this resource.”
  - Could be because the resource is in the cache already, or client is too busy, or whatever the application demands

The lifecycle of a stream is shown in Figure 2.



send: endpoint sends this frame
recv: endpoint receives this frame

H: HEADERS frame (with implied CONTINUATIONs)
PP: PUSH_PROMISE frame (with implied CONTINUATIONs)
ES: END_STREAM flag
R: RST_STREAM frame

HTTP/2

Design Goals

✅ **Eliminate HoL blocking by multiplexing HTTP requests over a single TCP connection**

✅ **Give servers more agency (e.g., allow them to *push* content over persistent connections)**

3. Reduce unnecessary duplicate bytes sent over the wire (e.g., static headers)

HTTP/2

Goal 3: Remove duplicate information as much as possible

- In HTTP/1.x, headers are always sent as plain text, despite the fact that many are static and unchanging
  - We already compress application data (e.g., with Content-Encoding: **gzip**), but we don’t do this for headers @ the protocol level
- HTTP/2 solves this with a new compression algorithm, HPACK, which has two main ideas
  - Compress header data (Huffman coding)
  - Keep a shared compression table on the client + server that is dynamically updated with new requests every on every request / response

HTTP/2

HPACK Compression Table

- HPACK encodes a static table with 61 entries for the most common HTTP headers (and some other freebies, like GET, POST) into every client and server
  - You no longer have to send these headers in cleartext, you can just send the encoded value of the index instead
- After this, every subsequent request is dynamically encoded and added to the shared table, which reduces the amount of data required to be sent over the wire for subsequent requests

Index	Header Name	Header Value
1	:authority	
2	:method	GET
3	:method	POST
…		
28	content-length	
38	host	
61	www-authenticate	
62	Host	kumarde.com

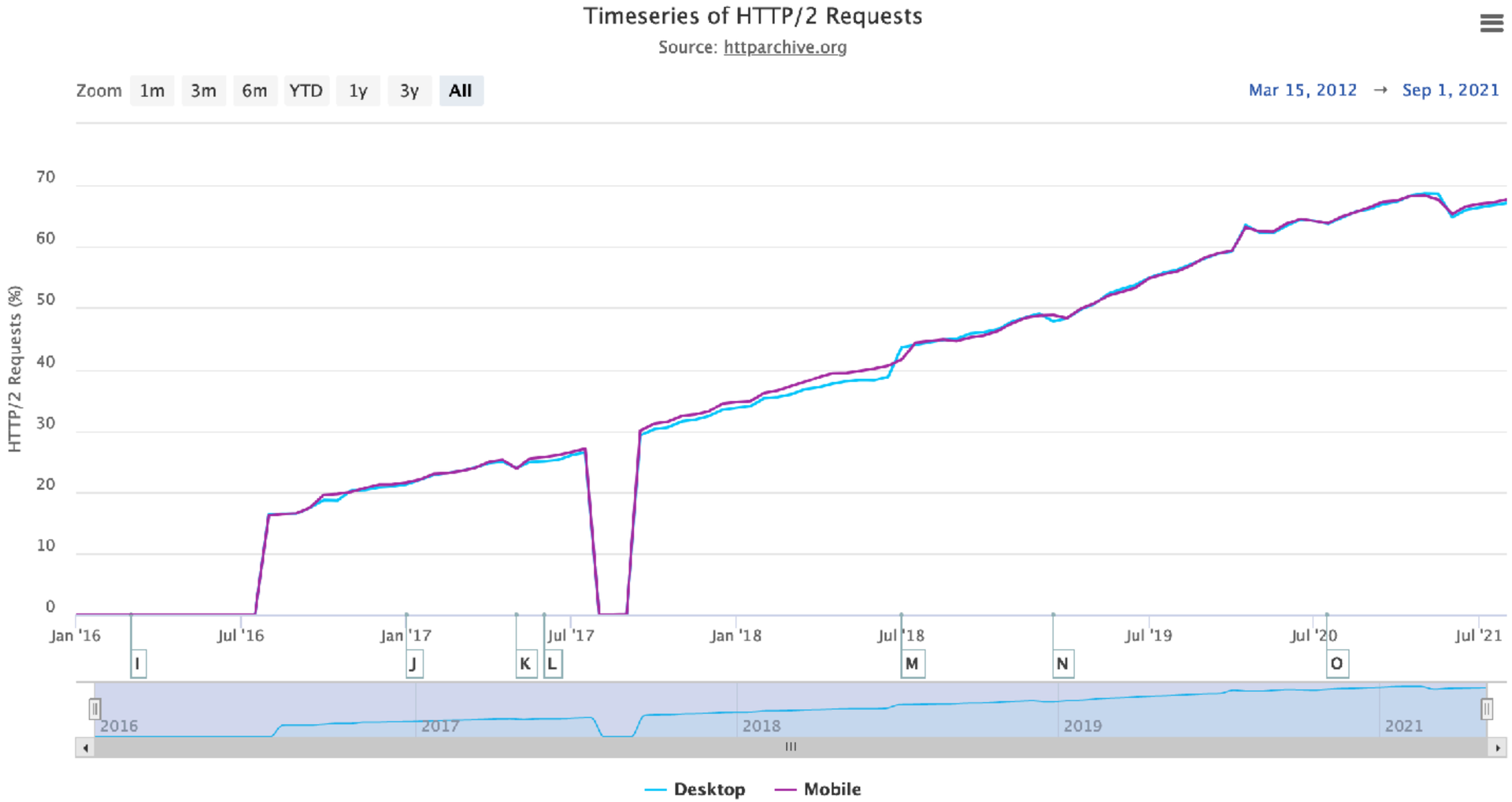
HTTP/2

Design Goals

- ✅ Eliminate HoL blocking by multiplexing HTTP requests over a single TCP connection
- ✅ Give servers more agency (e.g., allow them to *push* content over persistent connections)
- ✅ Reduce unnecessary duplicate bytes sent over the wire (e.g., static headers)

HTTP/2

Adoption is booming



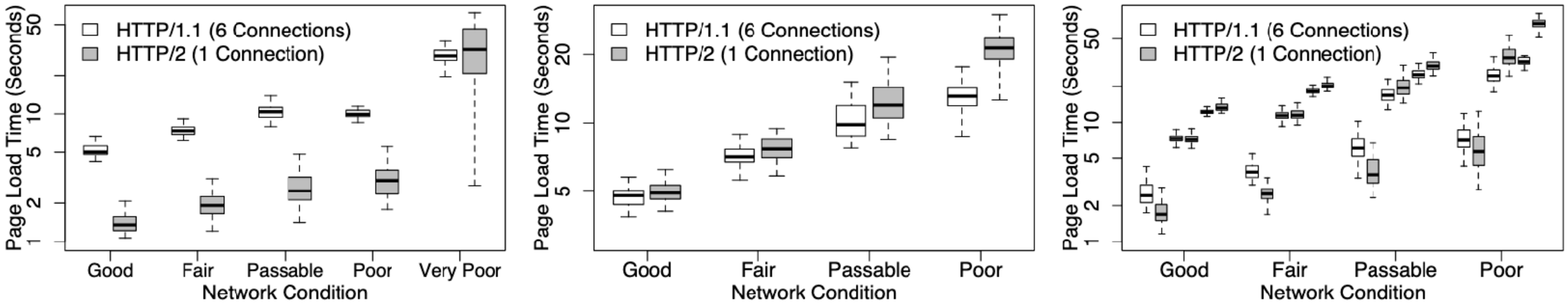
HTTP/2

Does it work?

- Generally, HTTP/2 will show performance benefits over HTTP/1.1 for well-resourced, high bandwidth channels
  - Financial Times reported speedups of 25 – 50% in a direct comparison between HTTP/1.x and HTTP/2
- But turns out this isn’t universally true…

HTTP/2

Does it work?



(a) 365 objects of 2 KB each (b) 10 objects of 435 KB each (c) 136 objects of 1 KB to 620 KB each

Figure 1: Distribution of page load times when loaded over `h1` and `h2` under various network conditions.

- "HTTP/2 Performance in Cellular Networks", from Montana State + Akamai, showed that in poor network conditions, HTTP/2 performed **worse** than HTTP/1.1, especially for larger objects. **Why?**

HTTP/2

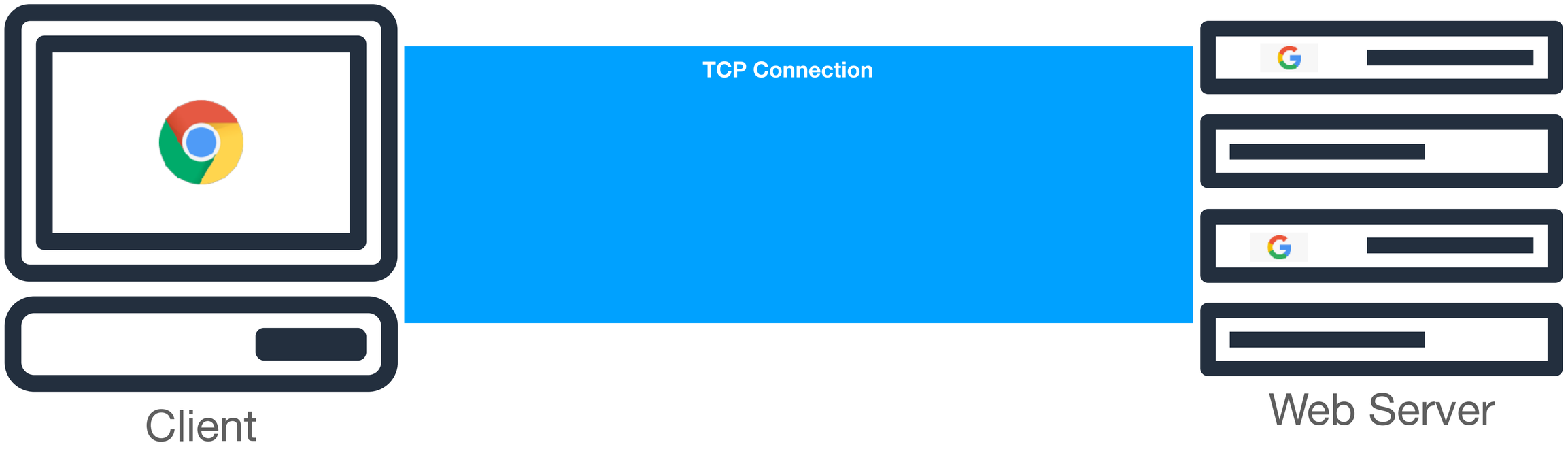
A New Problem

- HTTP/2 solves the HTTP-level HoL blocking problems associated with older versions of HTTP… but introduces a new problem at a **lower layer**

HTTP/2

A New Problem

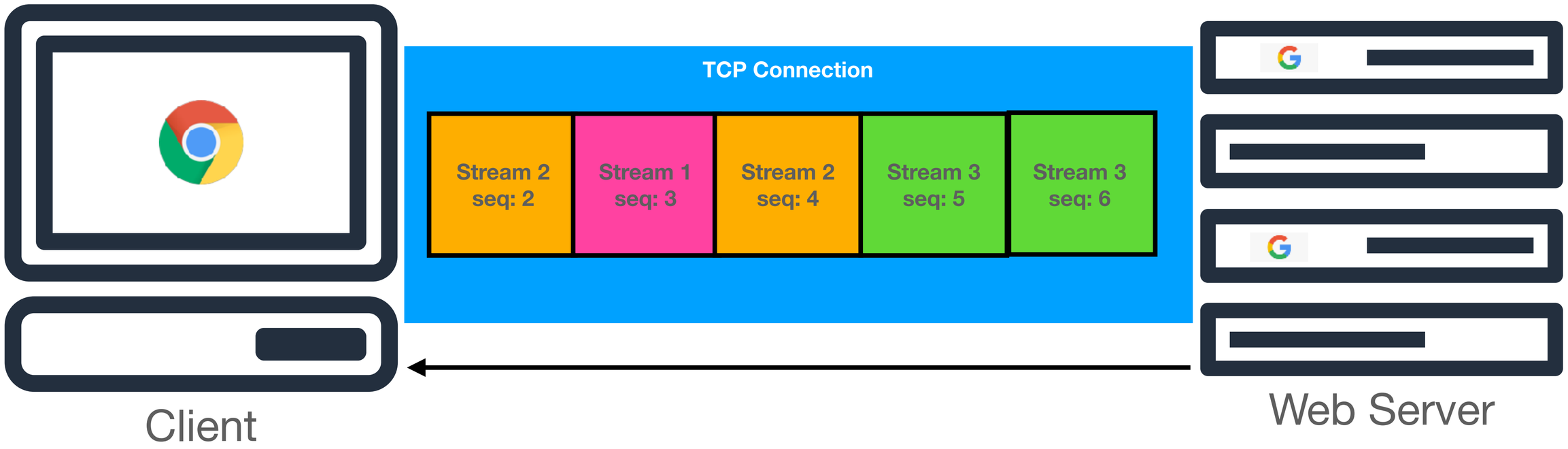
- HTTP/2 solves the HTTP-level HoL blocking problems associated with older versions of HTTP… but introduces a new problem at a **lower layer**



HTTP/2

A New Problem

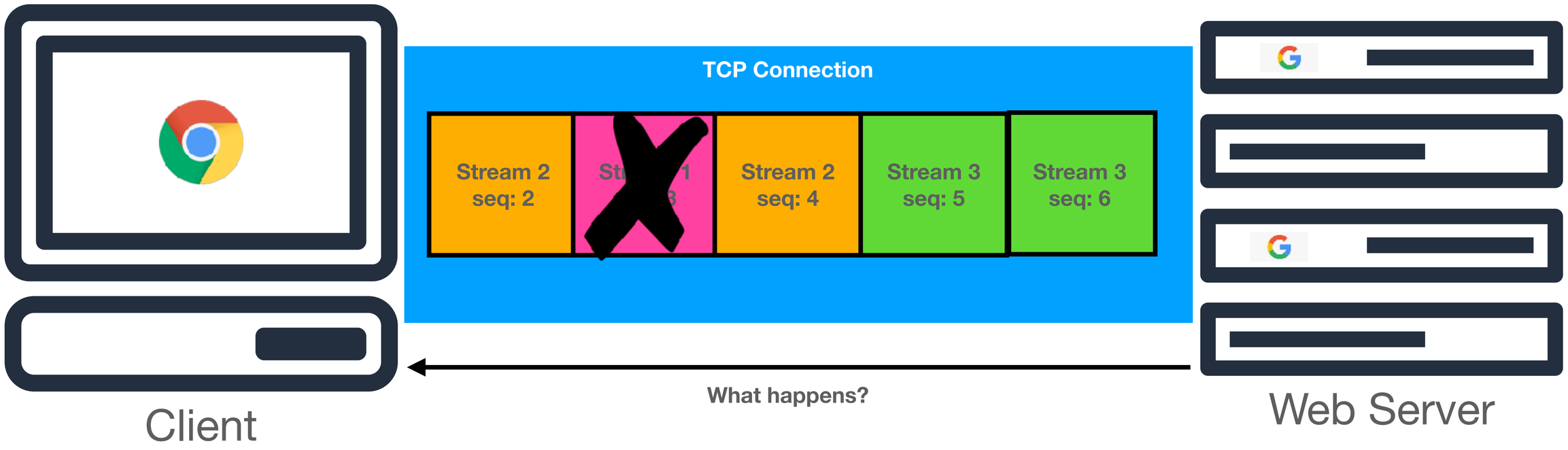
- HTTP/2 solves the HTTP-level HoL blocking problems associated with older versions of HTTP… but introduces a new problem at a **lower layer**



HTTP/2

A New Problem

- HTTP/2 solves the HTTP-level HoL blocking problems associated with older versions of HTTP… but introduces a new problem at a **lower layer**

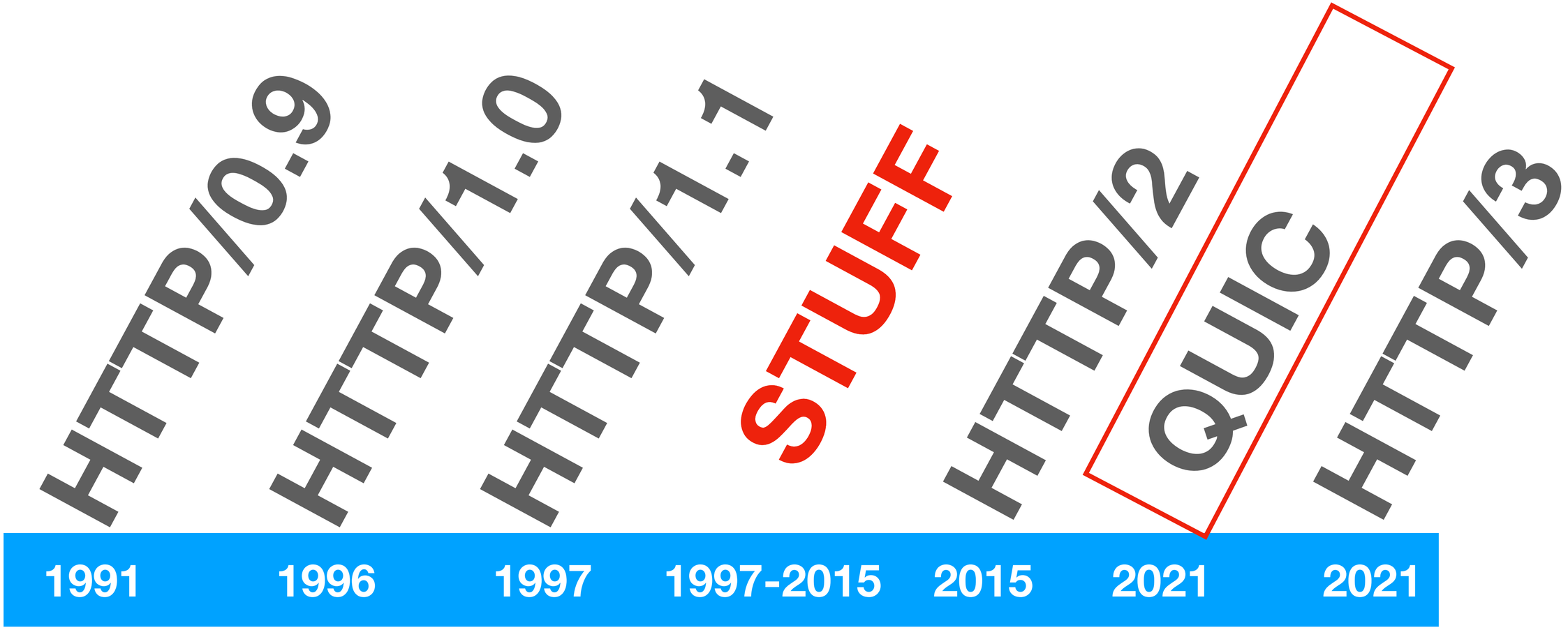


HTTP/2

A New Problem

- HTTP/2 solves th iated with older versions of HTT em at a **lower layer**

A History of Web Protocols



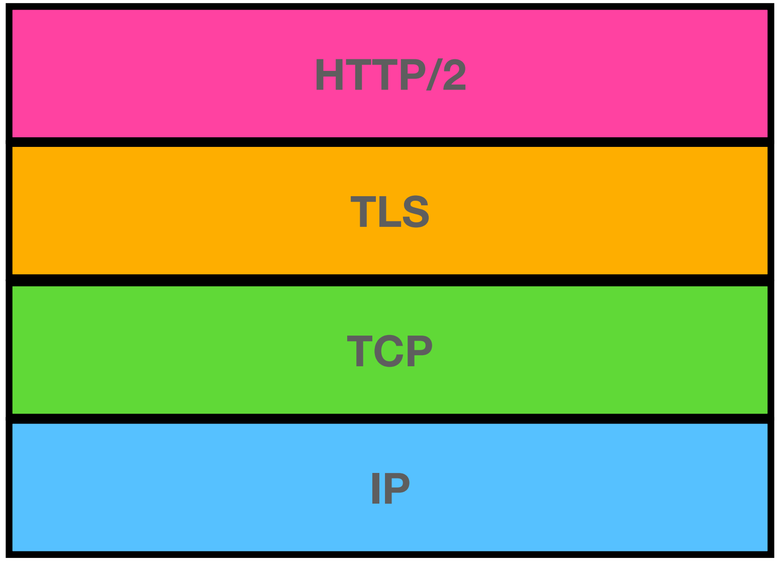
QUIC

A New Way Forward

- A core problem with HTTP up to this point is a fundamental limitation of *reliable transport*
  - We want to have reliability guarantees, but the way this is implemented in the layering model (e.g., in TCP) makes it such that applications don't have flexibility to define what reliability means!
- We could try to change TCP?
  - But that requires updating every router in the world. Way too hard.
- QUIC idea: What if we re-envisioned what we needed from lower network layers?

QUIC

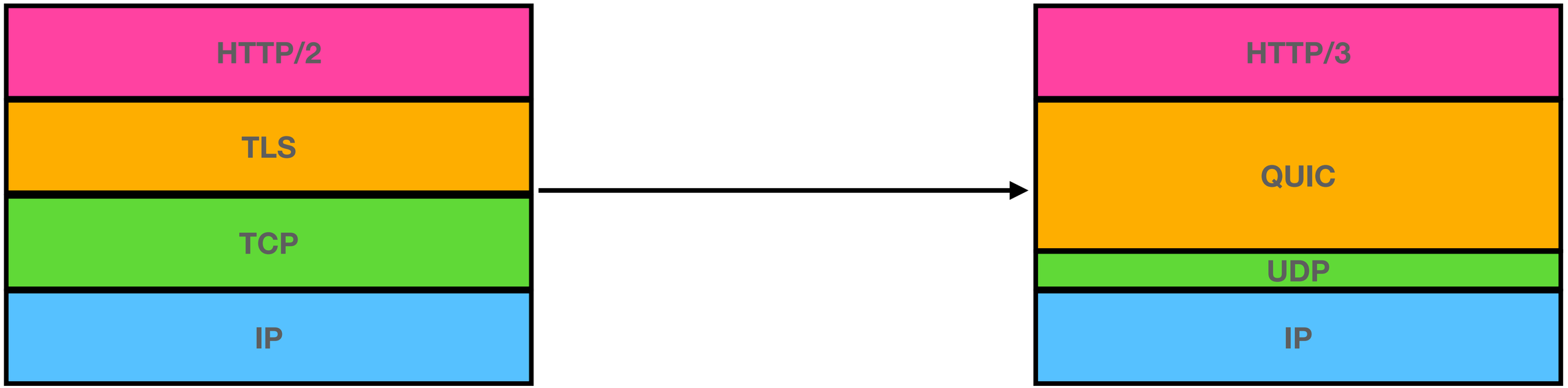
A New Transport Layer



The current world

QUIC

A New Transport Layer

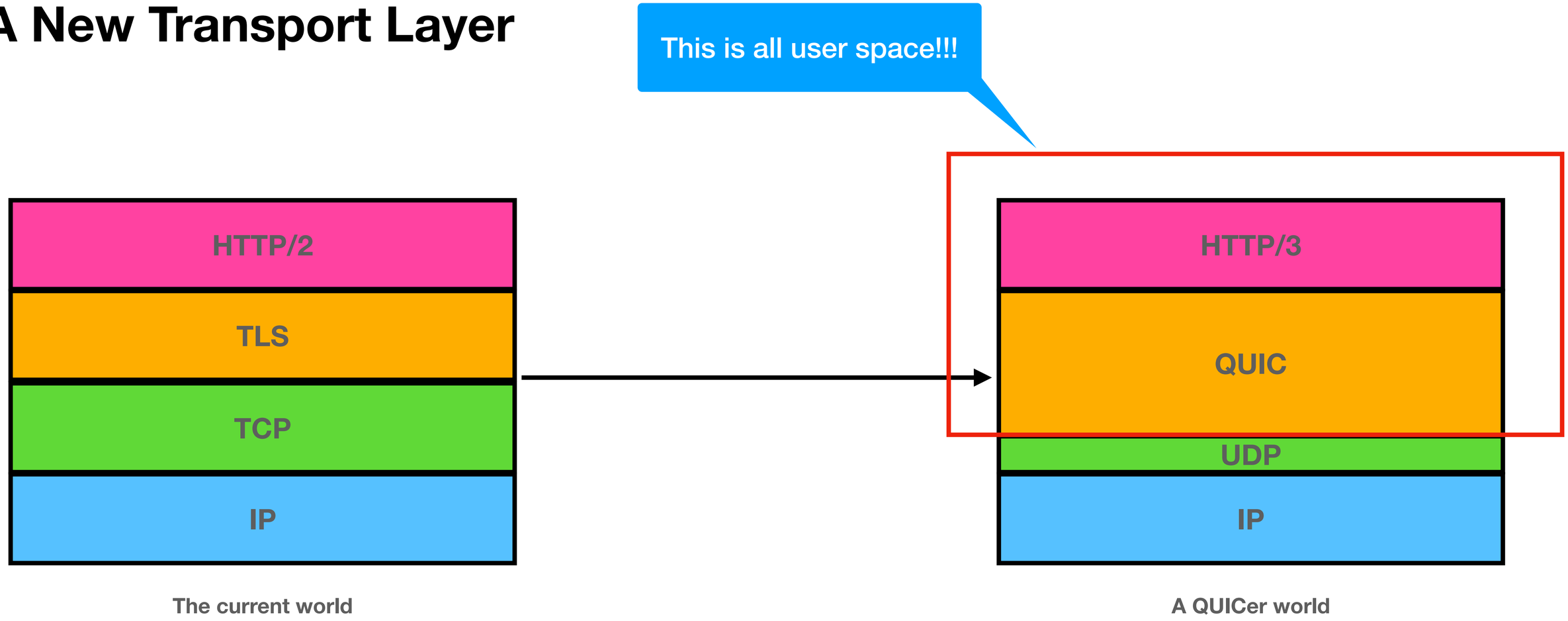


The current world

A QUICer world

QUIC

A New Transport Layer



QUIC

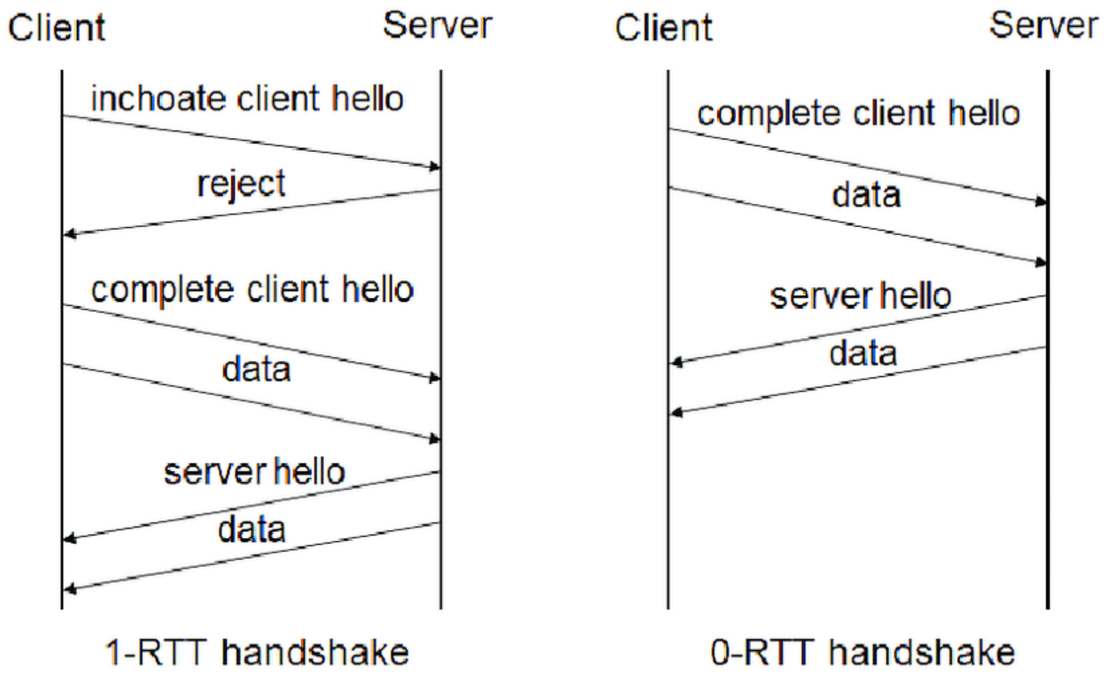
Design Goals

- A new, reliable transport layer
- Easily deployable and evolvable
  - Make this something that exists in userspace and something that doesn’t require us to update every router ever
- Security by default
  - Build in encryption, integrity checks, and authentication into the transport layer itself
- Reduce unnecessary delays imposed by strict layering
  - Handshake delays (e.g., TLS handshake), HoL blocking (HTTP, TCP)

QUIC

Establishing a Connection

- The first time a client wants to communicate with a server, it send an *inchoate client hello* in cleartext, which will initiate a REJ (reject) from the server
  - The server will send back a number of details, including a certificate chain (for server authentication) and other server metadata
- The client will then use the server information provided to send a *complete client hello*, and immediately start sending encrypted data
- Client *caches* server details (based on origin), so for any future connection, the client can simply use the server data to send encrypted messages moving forward. This is known as a **0-RTT protocol.**



QUIC

Two Types of Headers

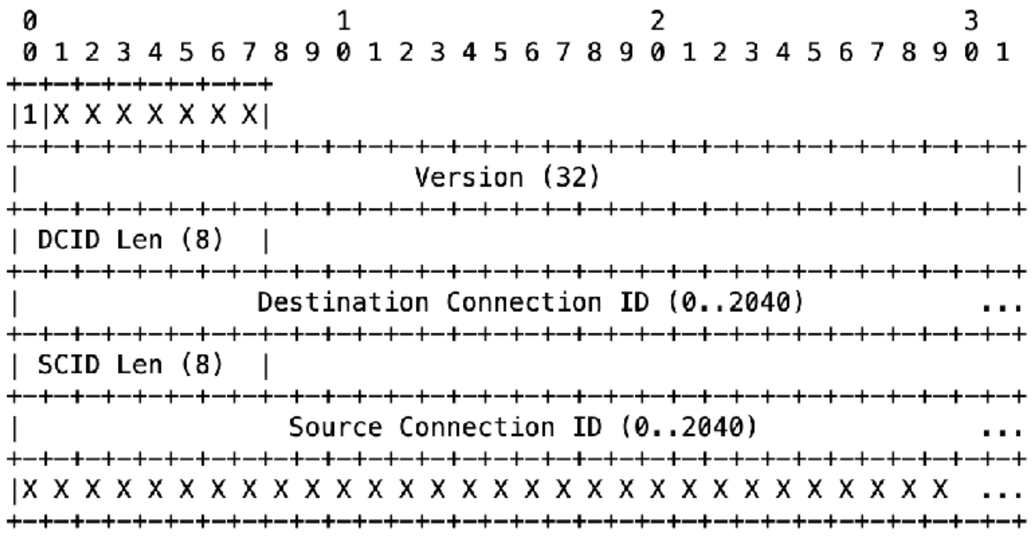
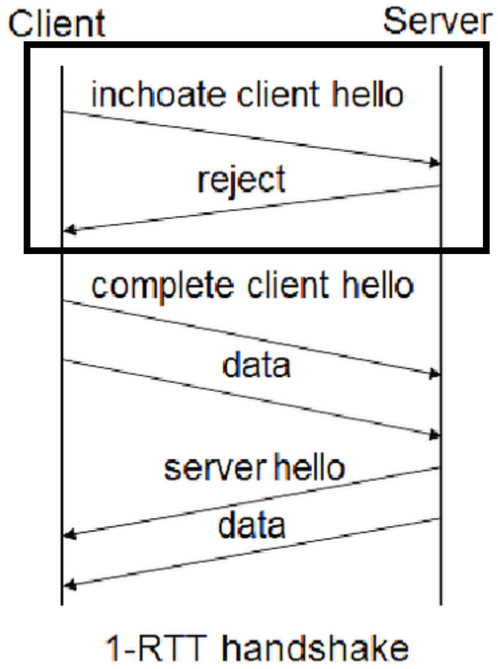
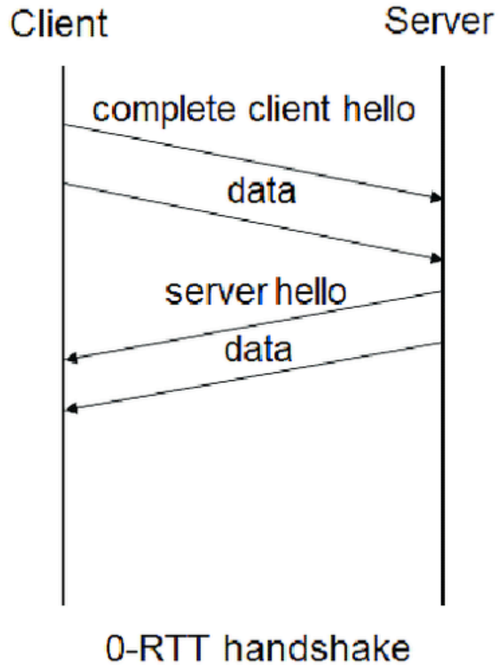


Figure 1: QUIC Long Header



1-RTT handshake



0-RTT handshake

https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-19#section-17.2

QUIC

Two Types of Headers

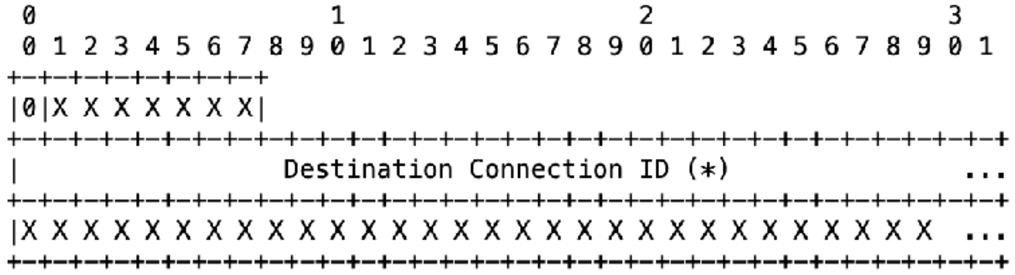
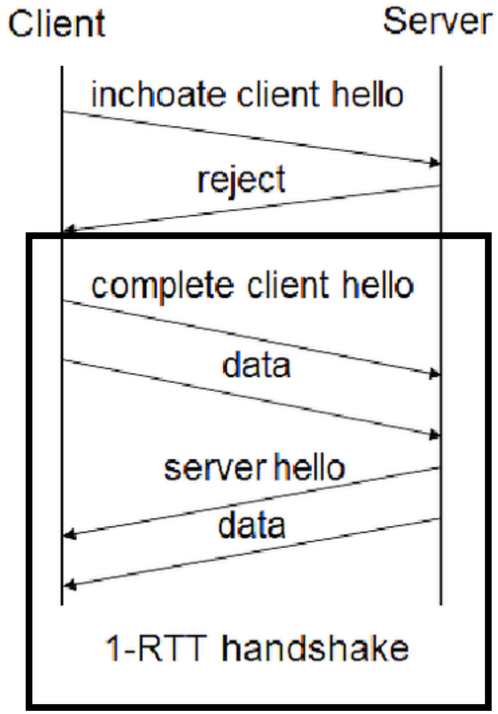
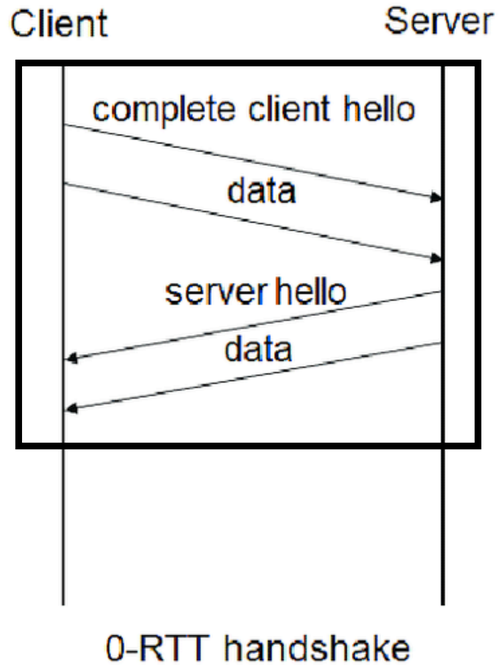


Figure 2: QUIC Short Header



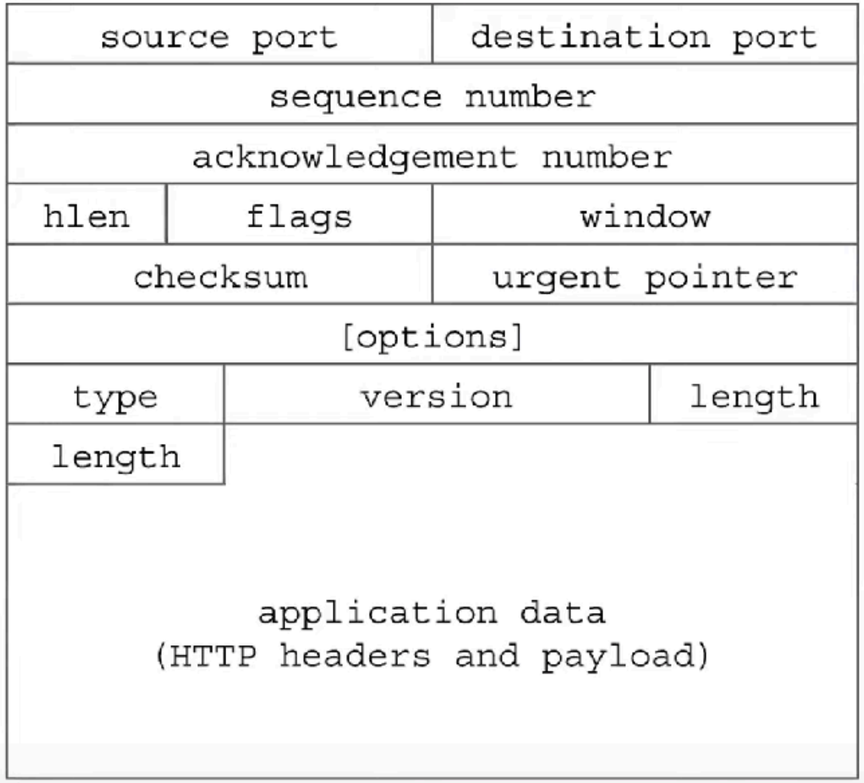


https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-19#section-17.2

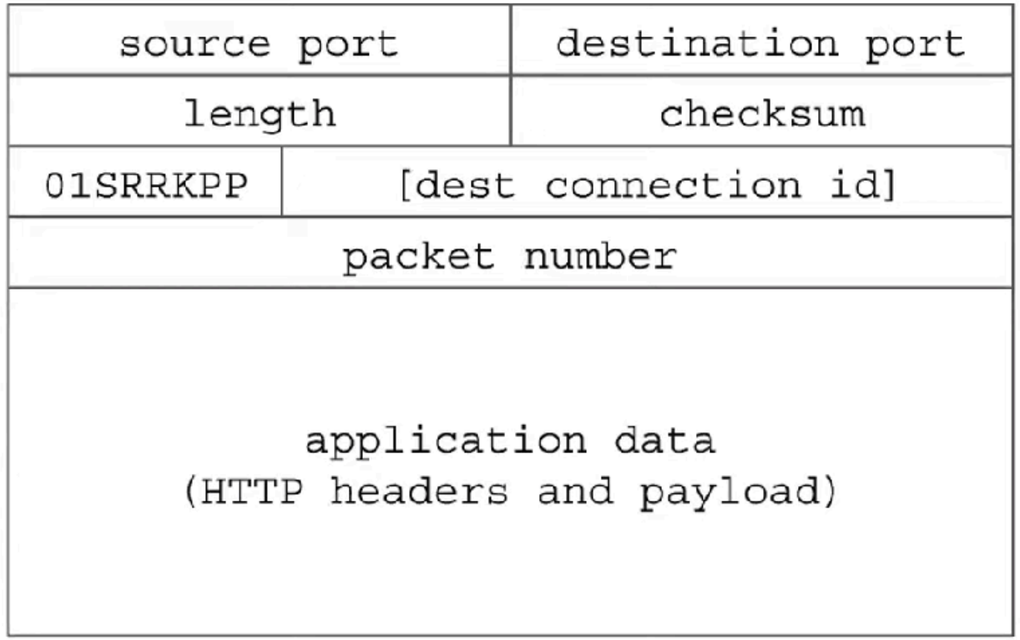
QUIC

Encrypt as much as possible

HTTP w/ TLS + TCP



HTTP w/ QUIC

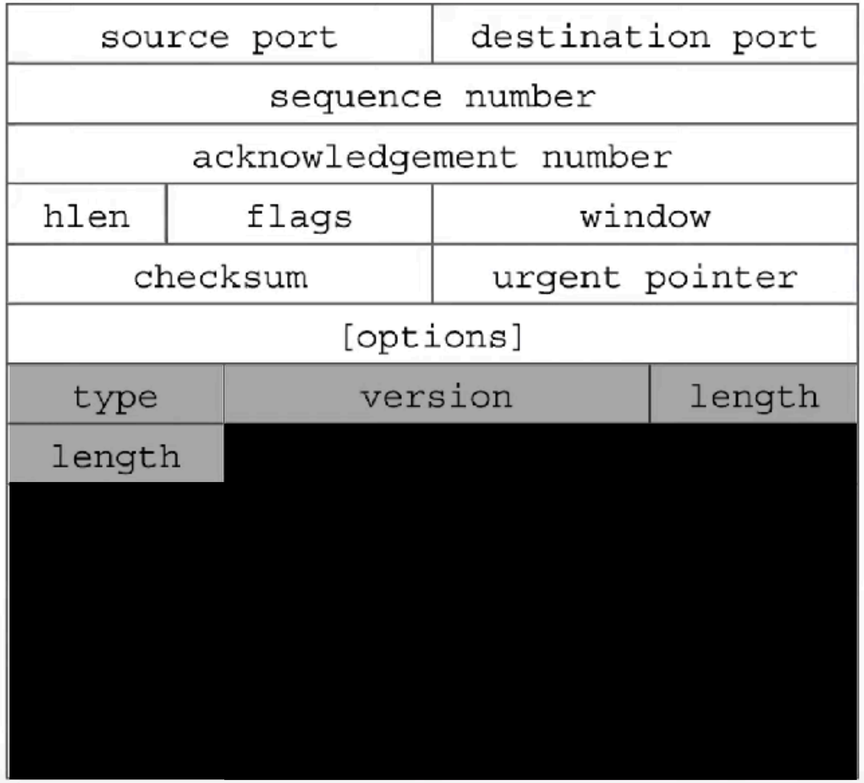


Slide stolen from: https://www.youtube.com/watch?v=31J8PoLW9iM&t=9104s

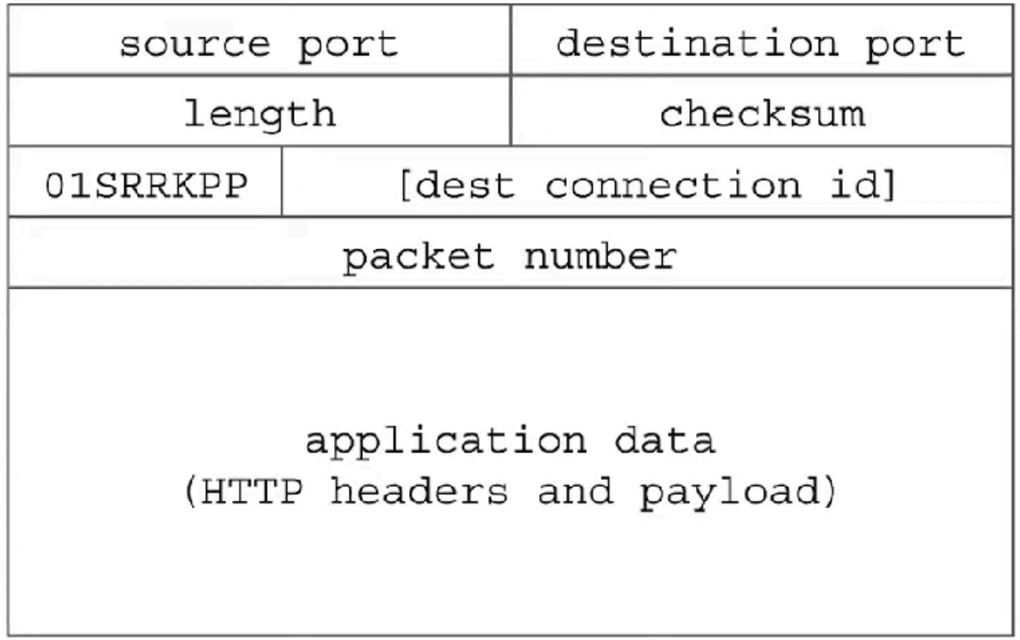
QUIC

Encrypt as much as possible

HTTP w/ TLS + TCP



HTTP w/ QUIC

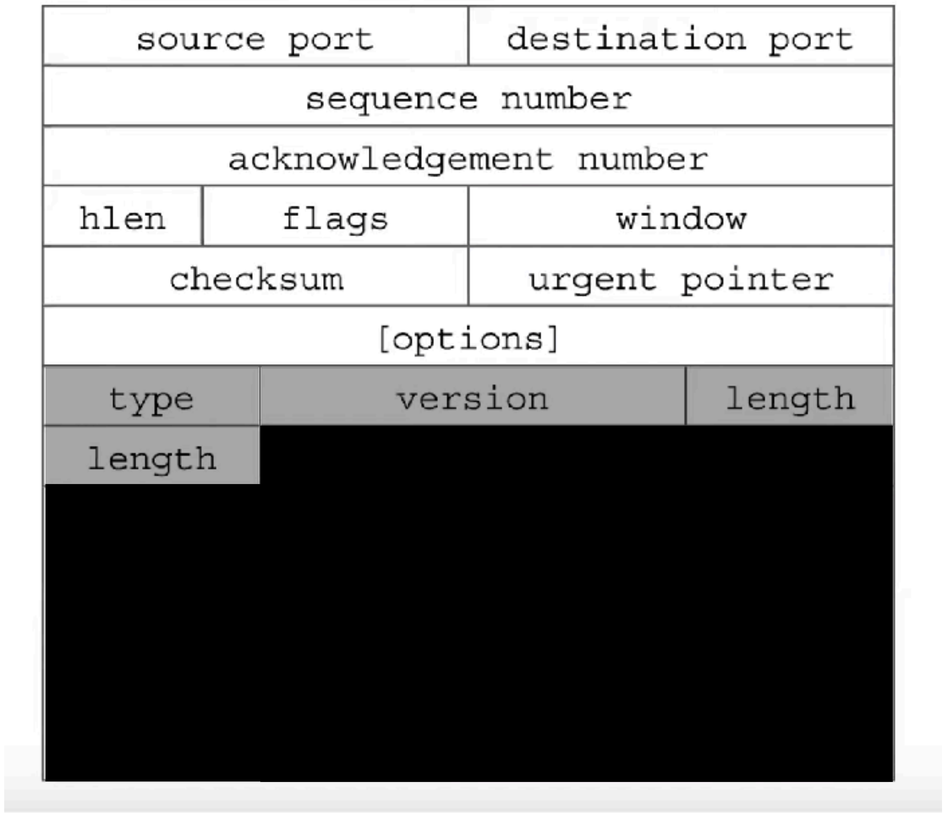


Slide stolen from: https://www.youtube.com/watch?v=31J8PoLW9iM&t=9104s

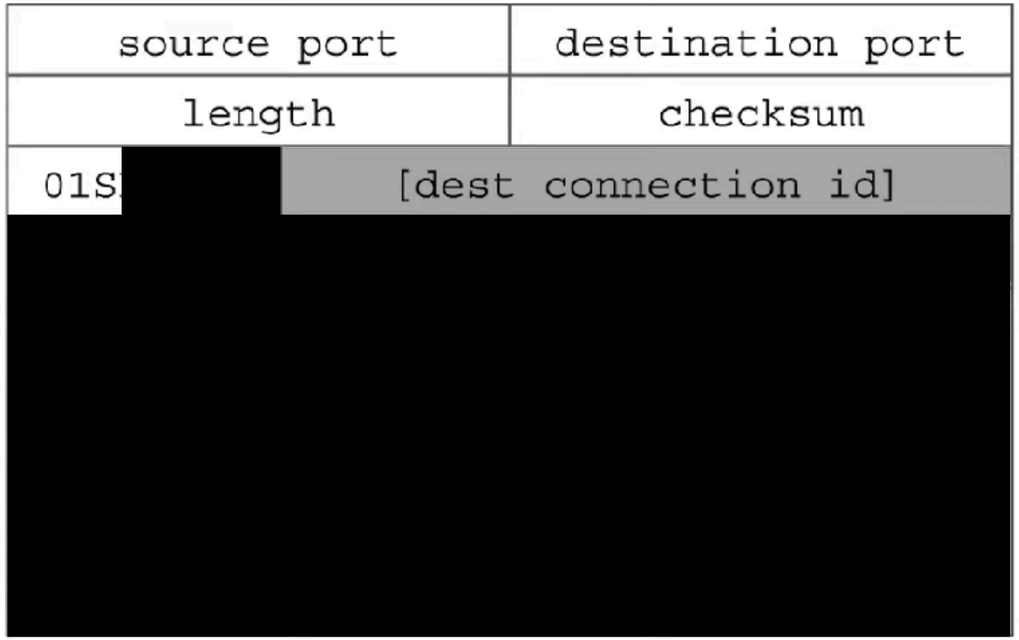
QUIC

Encrypt as much as possible

HTTP w/ TLS + TCP



HTTP w/ QUIC

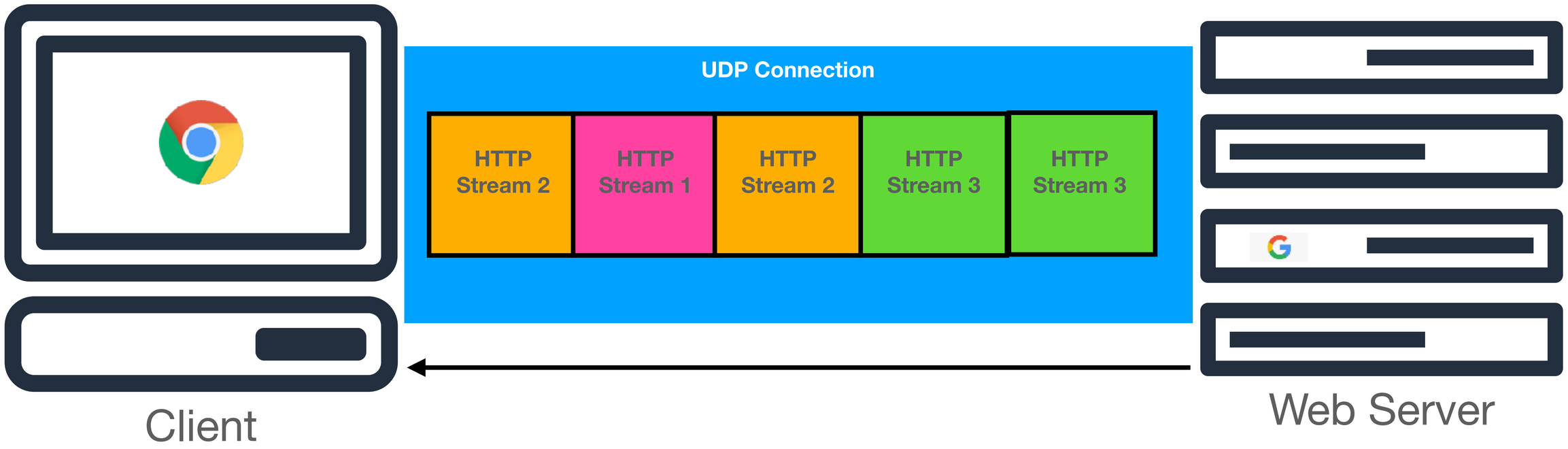


Slide stolen from: https://www.youtube.com/watch?v=31J8PoLW9iM&t=9104s

QUIC

Maintaining the Stream Abstraction

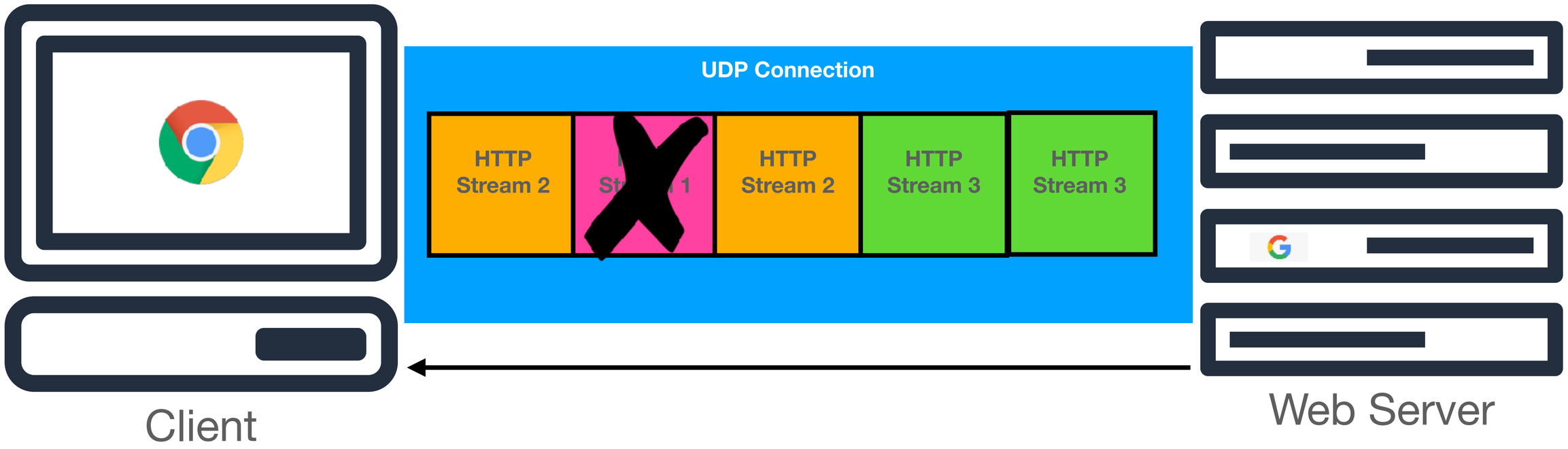
- QUIC uses the idea of a stream (with a stream_id) as a baseline abstraction for sending data between two endpoints, similar to HTTP/2



QUIC

Maintaining the Stream Abstraction

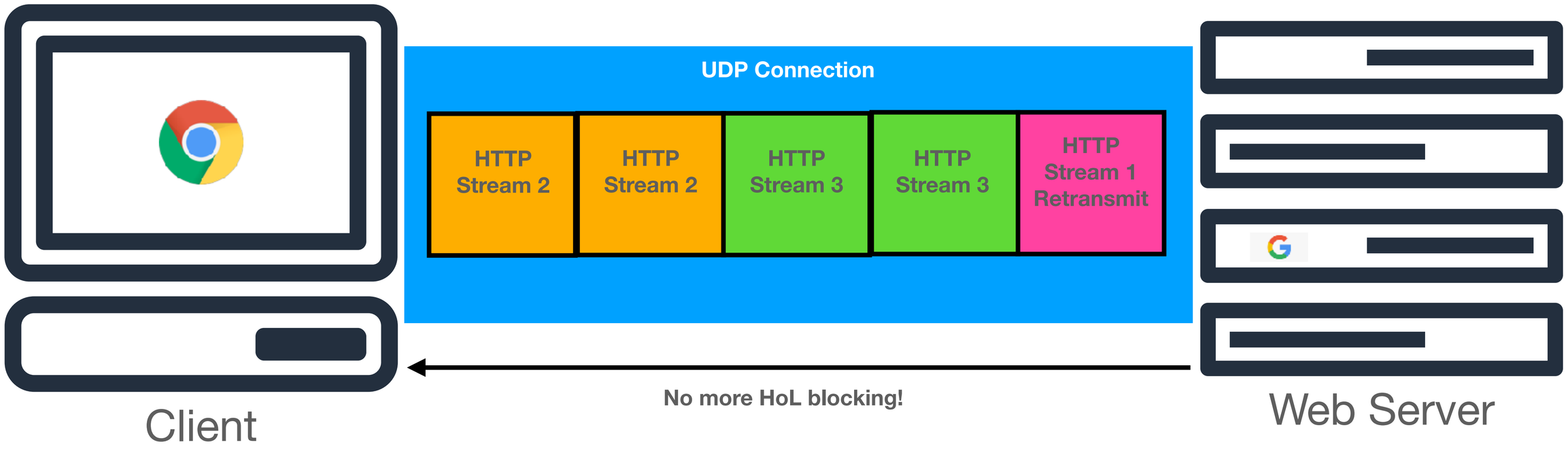
- QUIC uses the idea of a stream (with a stream_id) as a baseline abstraction for sending data between two endpoints, similar to HTTP/2



QUIC

Maintaining the Stream Abstraction

- QUIC uses the idea of a stream (with a stream_id) as a baseline abstraction for sending data between two endpoints, similar to HTTP/2



TCP vs. QUIC

Recovering from Losses

- TCP uses sequence numbers + acknowledgement numbers to identify whether or not a packet has been lost, and needs to be retransmitted
  - Unfortunately, sequence numbers mean two things: reliability **and** the order at which the bytes are supposed to be delivered to the receiver
  - On top of this, TCP retransmissions use the *same* sequence number, so it becomes very hard to know whether an ACK was sent for first transmission or a retransmission
  - TCP conflates transmission ordering AND delivery ordering in one number

TCP vs. QUIC

Recovering from Losses

- QUIC decouples transmission and delivery ordering through its use of *streams*
  - Each packet contains a packet number, which is **unique and monotonically increasing, even on retransmission**
  - Clients will ACKNOWLEDGE packet numbers, and the server can identify if an outstanding packet has not been acknowledged… you can find the details at the link below
  - Each frame in a stream contains a *stream offset*, which alerts the client of how to properly reorder the packets on the delivery side
- Enables simpler loss detection than TCP

https://tools.ietf.org/id/draft-ietf-quic-recovery-27.html

QUIC

Connection Rebinding

- Because QUIC connections are over UDP, they can persist *beyond traditional network boundaries*, like your home NAT
  - No more resetting connection when your underlying network changes
- QUIC does this through the use of several unique variable length Connection IDs to identify the connection, with a protocol in place to verify the connection through a network change
- See RFC for notes on address spoofing + off-path packet attackers (something they've considered!)

QUIC

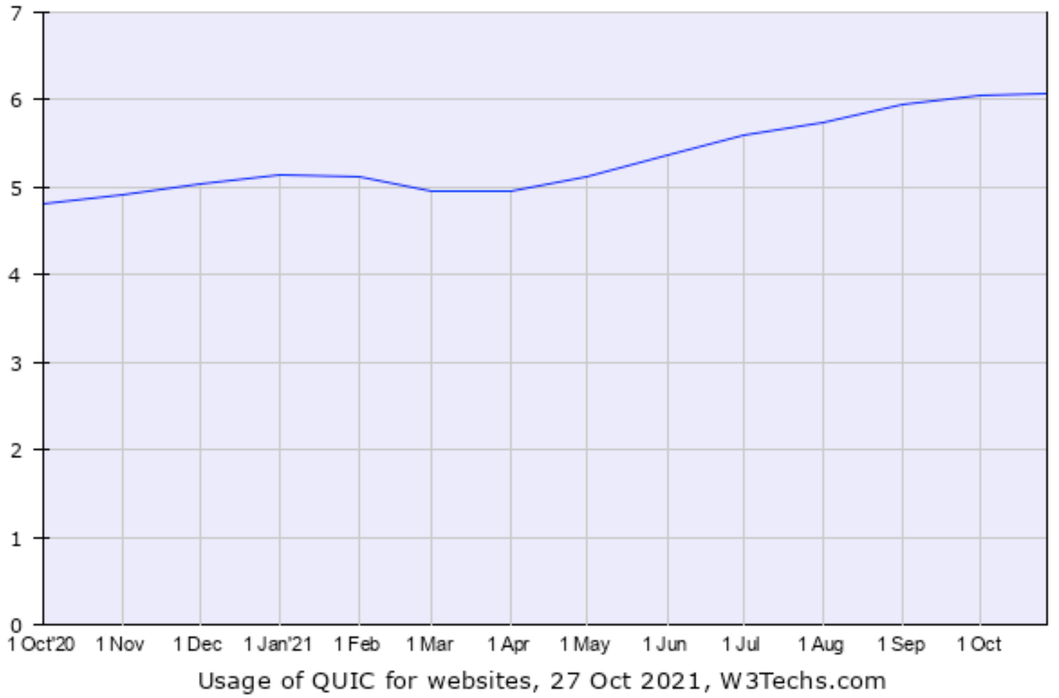
NATs, Middleboxes, Deployment Challenges

- Typically, NATs keep track of TCP connections by using a 5-tuple (src_port, src_ip, dst_port, dst_ip, protocol), and can maintain state because they have access to TCP headers
- Not all NATs speak QUIC yet, and even if they did, header information is encrypted, so they default to processing UDP packets, which could cause short timeouts and routing issues
- UDP-based protocols are susceptible to *reflection attacks,* where attackers use UDP servers with spoofed source ports to amplify their attack, and QUIC can be asymmetric on *inchoate client hello*
    - This is why QUIC has a REJ packet to start, but this increases the number of round trips required on initial connection. Probably a decent trade off.

QUIC Deployment

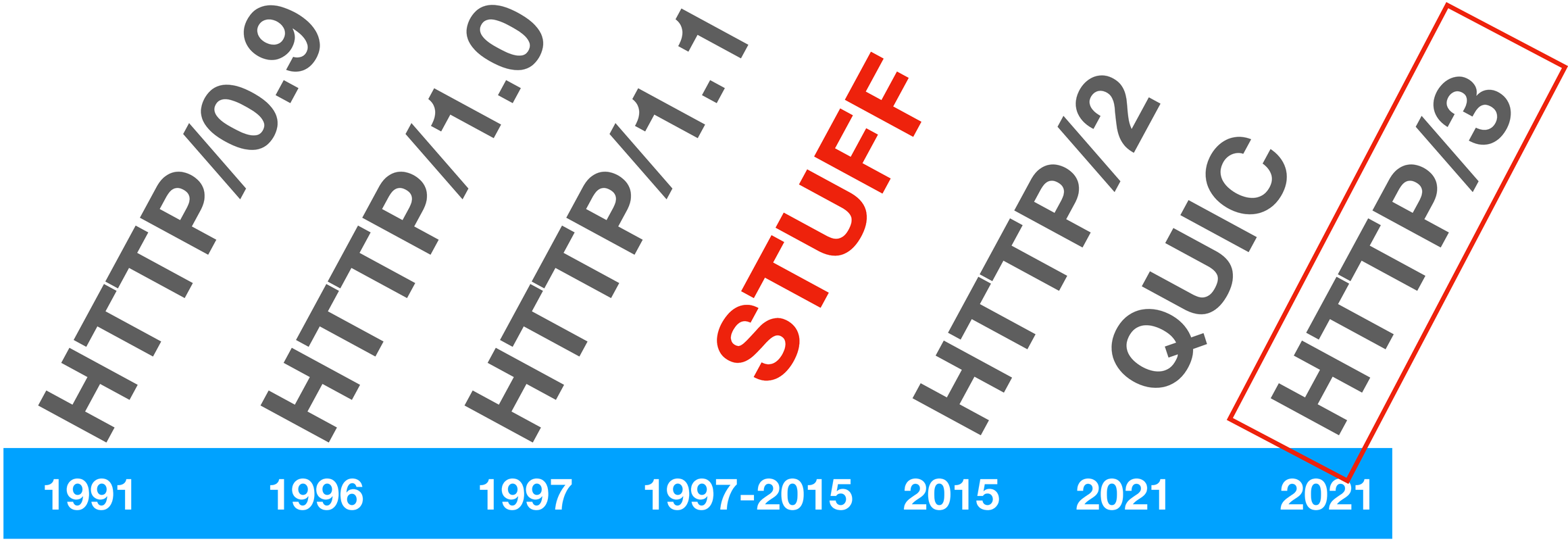
QUICly eating the world

- QUIC was officially ratified by the IETF in May 2021 (RFC 9000)
- QUIC support already existed in Chrome for a while, but is now available in Firefox as well
- QUIC is being deployed everywhere
  - 6% of websites use QUIC, but will grow post RFC ratification
  - Google apps all use QUIC, 75% of Facebook uses QUIC
  - Some ISPs have reported that **20% of their packets were over QUIC**



https://w3techs.com/technologies/details/ce-quic

A History of Web Protocols



HTTP/3 is HTTP over QUIC!

Recap

- The web has drastically changed over time, with developers doing more than ever before and websites becoming increasingly complex
  - But for a long time, our protocols didn’t match the growing complexity of the world
- New protocols like SPDY, HTTP/2 were useful in working within our paradigm, but there is **change** afoot!
  - People are not liking TCP as much, and companies like Google are starting to throw their weight around in envisioning a new future for layering requirements
- We are redefining “end-to-end” abstractions… let’s see how it goes :)