Computer Networking

COMP 177 | Fall 2020 | University of the Pacific | Jeff Shafer

HTTP/3

(also: HTTP/2, SPDY, QUIC)

HTTP/2 (and "Google SPDY")

Timeline

- **HTTP/0.9 (*not* an RFC: https://www.w3.org/Protocols/HTTP/AsImplemented.html)**
  - Initial version of HTTP — a simple client-server, request-response, telnet-friendly protocol
  - Request nature: single-line (method + path for requested document)
  - Methods supported: `GET` only
  - Response type: hypertext only
  - Connection nature: terminated immediately after the response
  - No HTTP headers (cannot transfer other content type files), No status/error codes, No URLs, No versioning

https://medium.com/platform-engineer/evolution-of-http-69cfe6531ba0

Timeline

- **HTTP/1.0 – RFC 1945 (May 1996)**
  - Browser-friendly protocol
  - Provided header fields including rich metadata about both request and response (HTTP version number, status code, content type)
  - Response: not limited to hypertext (`Content-Type` header provided ability to transmit files other than plain HTML files — e.g. scripts, stylesheets, media)
  - Methods supported: `GET` , `HEAD` , `POST`
  - Connection nature: terminated immediately after the response

https://medium.com/platform-engineer/evolution-of-http-69cfe6531ba0

Timeline

- **HTTP/1.1 – RFC 2068 (January 1997)**
  - Performance optimizations and feature enhancements
    - Persistent and pipelined connections
    - Chunked transfers
    - Compression/decompression
    - Virtual hosting (a server with a single IP address hosting multiple domains)
  - Methods supported: `GET` , `HEAD` , `POST` , `PUT` , `DELETE` , `TRACE` , `OPTIONS`
  - Connection nature: long-lived

https://medium.com/platform-engineer/evolution-of-http-69cfe6531ba0


THE WORLD IS CHANGING

Motivations

- **What's wrong with HTTP/1.1?**
  - Complicated specification
  - Many options / incomplete implementations
  - **Performance limitations**
- **Why does it take so long to load a web page?**

http archive

REPORTS DISCUSS ABOUT

The **HTTP Archive** Tracks How the **Web is Built.**

We periodically crawl the top sites on the web and record detailed information about fetched resources, used web platform APIs and features, and execution traces of each page. We then crunch and analyze this data to identify trends — learn more about our methodology.

View Reports

Featured Report

State of the Web

This report captures a long view of the web, including the adoption of techniques for efficient network utilization and usage of web standards like HTTPS.

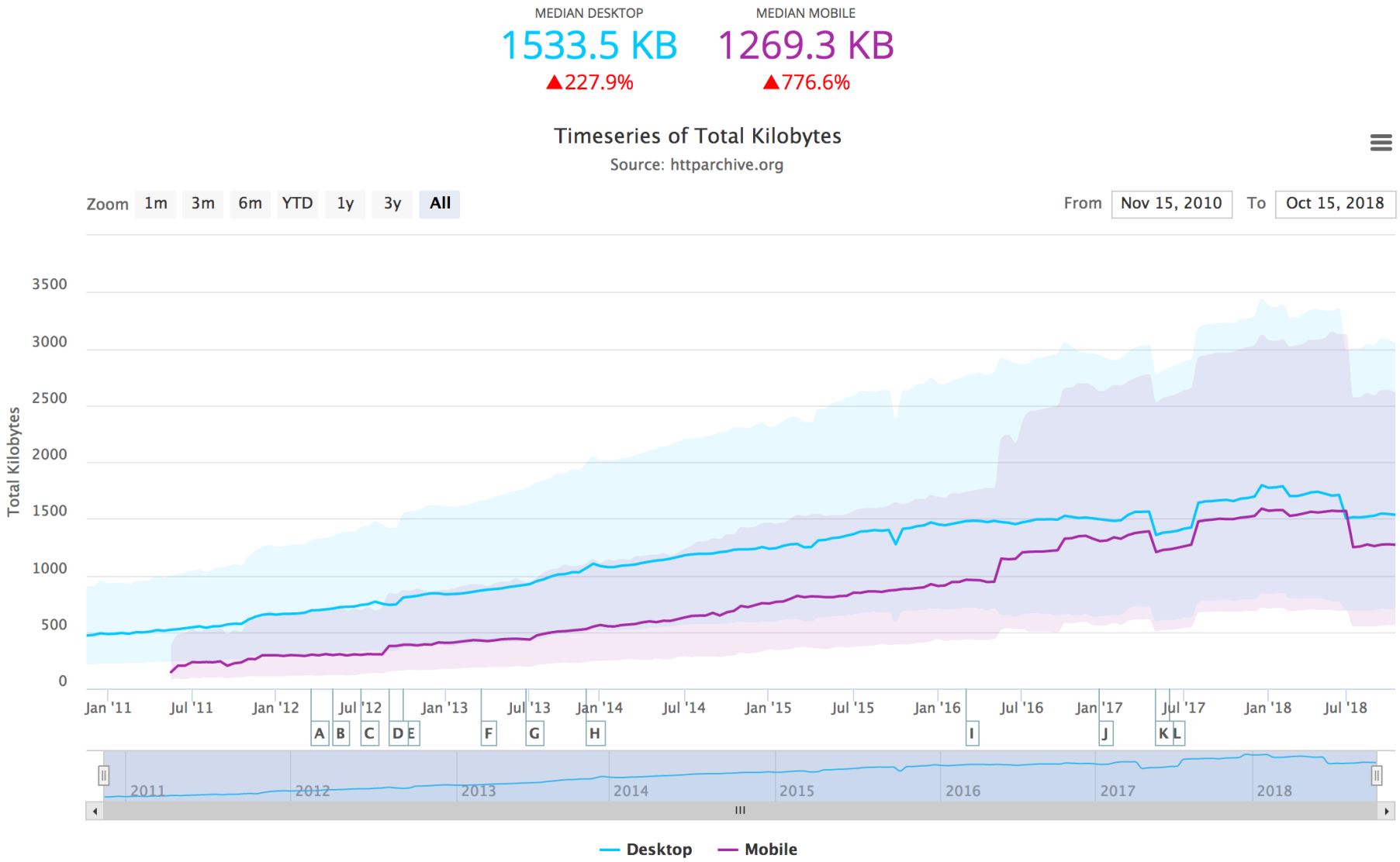
View The Report

https://httparchive.org/

Total Kilobytes

The sum of transfer size kilobytes of all resources requested by the page.

See also: Page Weight

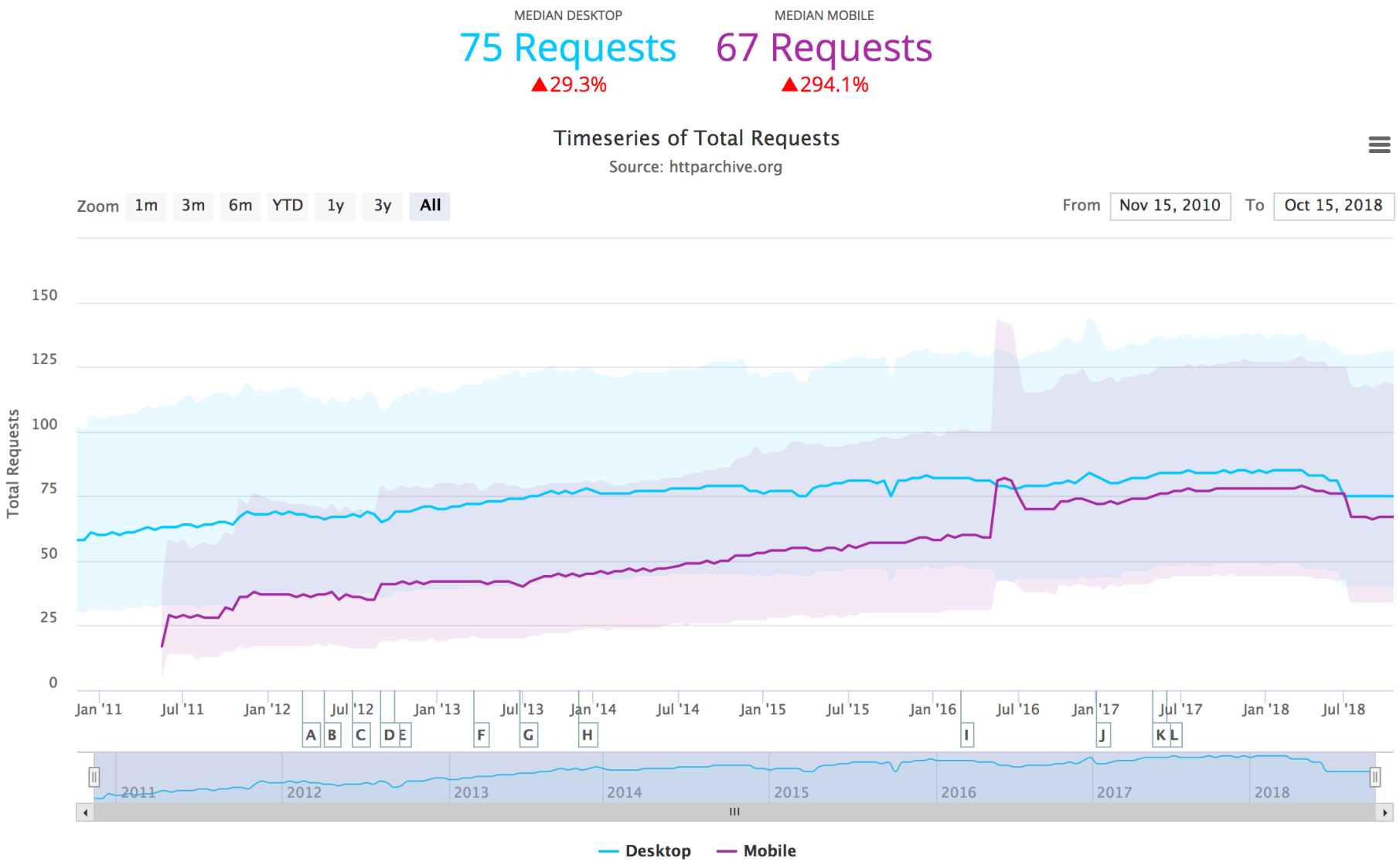


https://httparchive.org/reports/state-of-the-web

Total Requests

The number of resources requested by the page.

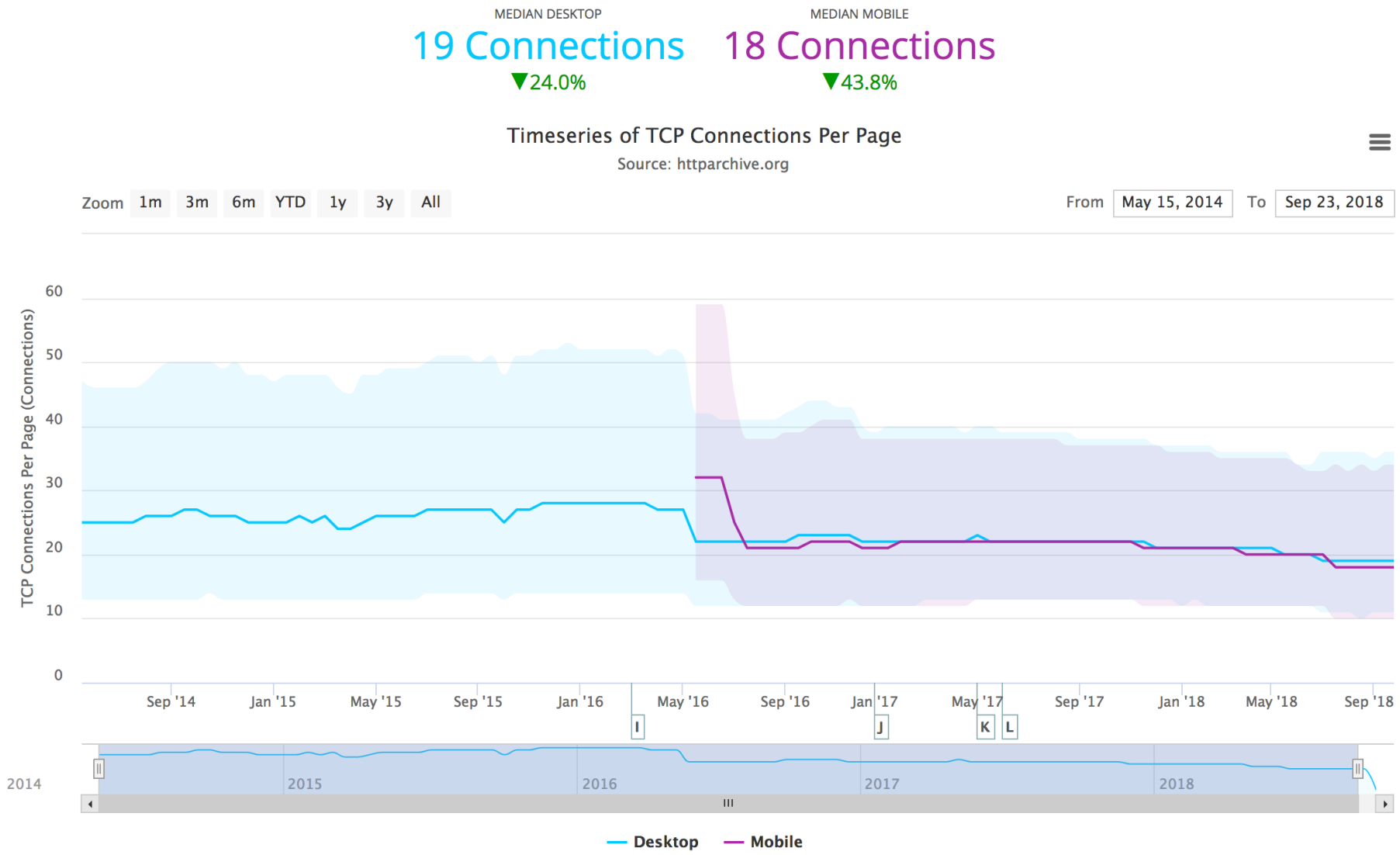
See also: Page Weight



https://httparchive.org/reports/state-of-the-web

TCP Connections Per Page

The number of TCP connections per page.



https://httparchive.org/reports/state-of-the-web

Performance

- Many files (Median of ~75 in 2018)
  - Some files are large, some files are small
- Many TCP sockets (Median of ~19 in 2018)
  - More than one file per socket
  - Resource penalty + time penalty to just open yet more sockets
- Problems
  - **Latency between requests**
    - HTTP Pipelining typically disabled by default
  - **Head-of-line blocking**
    - Multiple requests over same socket, large file blocks small file
    - **Not solved** by HTTP Pipelining

HTTP Performance Hacks

- **Sprite Sheets**
- Instead of sending many tiny images, send one big image and use JavaScript/CSS to pull out the desired pieces
- Inefficient if only a few pieces are needed

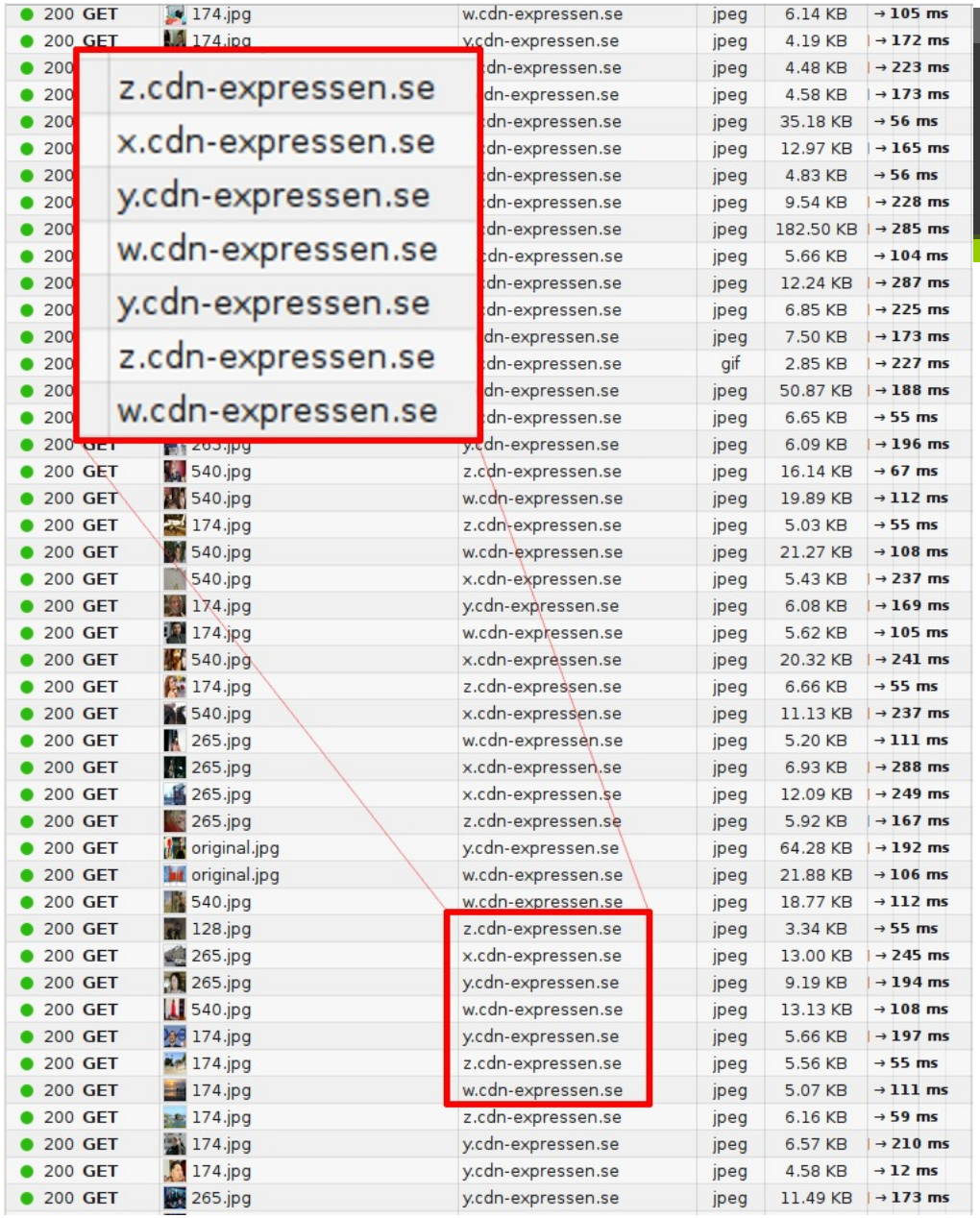
HTTP Performance Hacks

```
.icon1 {
  background:
url(<data>)
no-repeat;
}

.icon2 {
   background:
url(<data>)
no-repeat;
}
```

- **Inlining**
  - Embed data inside the CSS file instead of as separate files
- **Concatenation**
  - Combine multiple JS / CSS files into megafiles before sending to client

HTTP Hacks



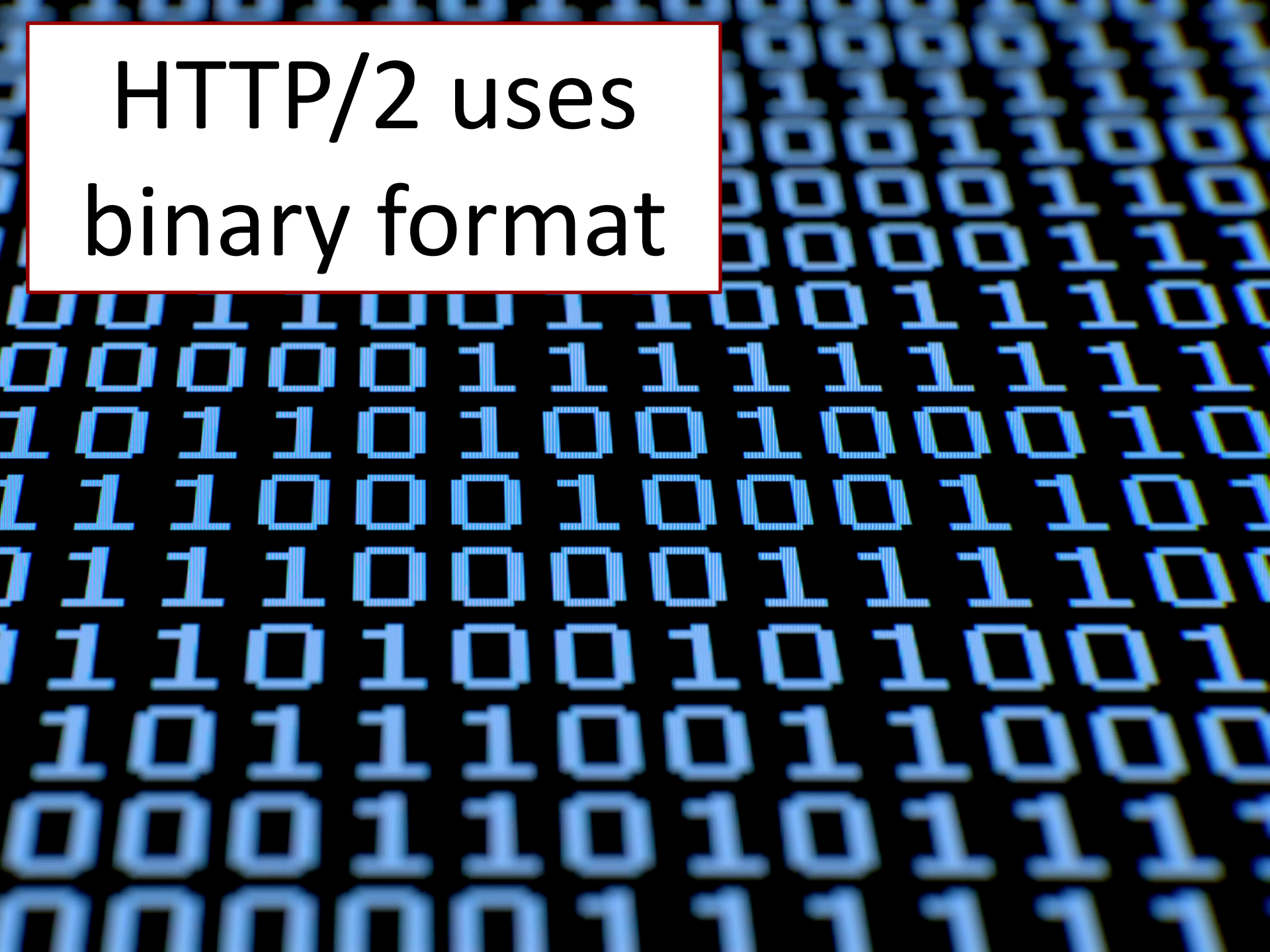
- **Domain Sharding**
- Avoid per-host limits on # of connections by spreading website across many "separate" hostnames
- How does the browser (or OS) prioritize these connections?

Timeline

- IETF HTTPbis working group
  - Formed in 2007
  - Finalized HTTP/2 – **RFC 7540 (May 2015)**
  - Finalize HPACK – **RFC 7541 (May 2015)**
  - Heavily derived from Google SPDY work
  - **https://http2.github.io/**
- Google SPDY
  - Released in Chrome in 2010
  - Withdrawn in Chrome in 2016

HTTP/2

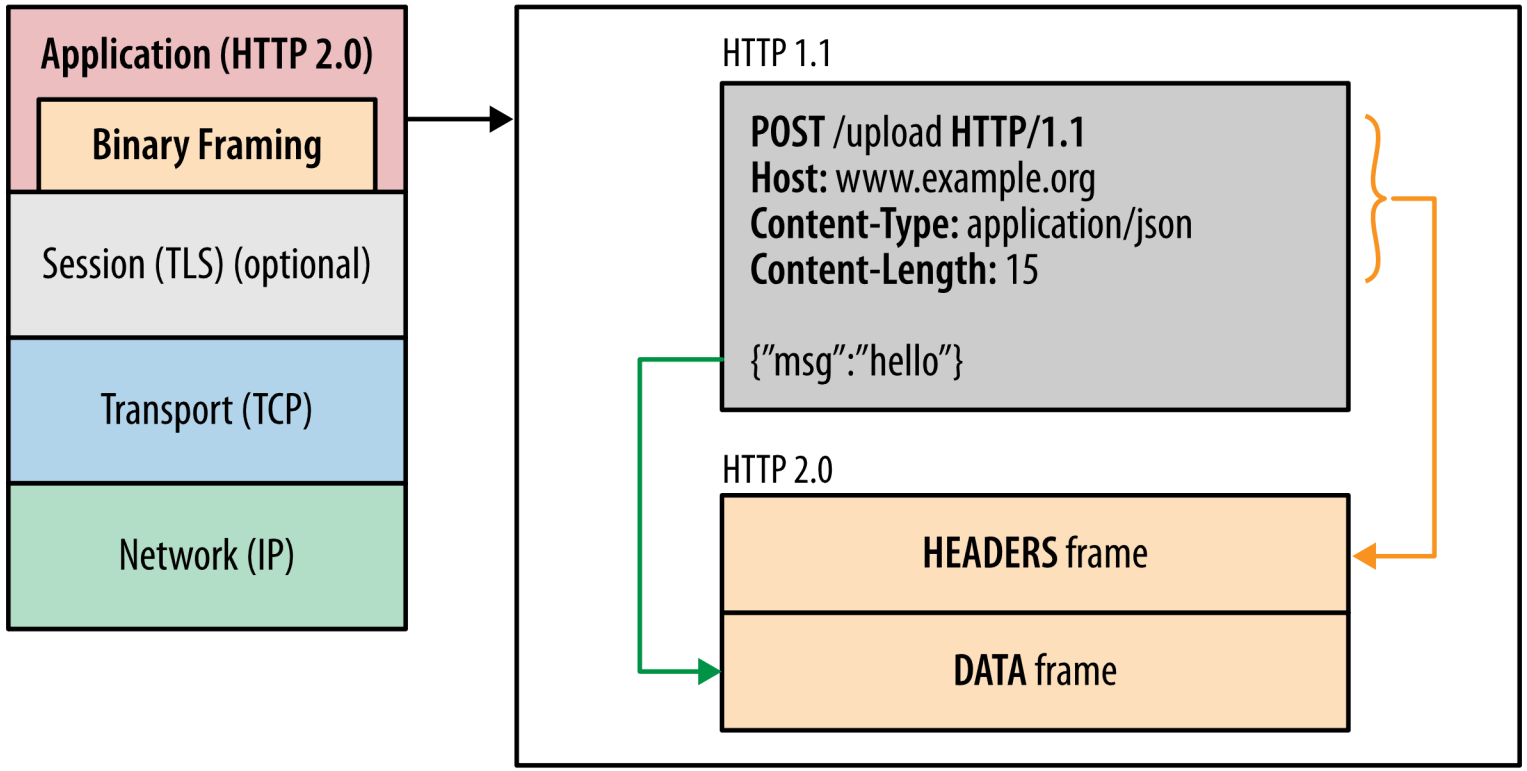
- Make the web faster *and* eliminate the design hacks
- Maintain high-level compatibility with HTTP/1.1
  - Methods, status codes, URIs, …
  - No changes to web pages, web apps, …
- Reduce Latency -> Reduce Page Load Times
  - Compress headers
  - ~~Server Push~~
    - Upcoming removal from Chrome – now considered more efficient for client to request resources it needs *after* checking the local cache: https://groups.google.com/a/chromium.org/g/blink-dev/c/K3rYLvmQUBY/m/vOWBKZGoAQAJ?pli=1
  - No head-of-line blocking *(not fully successful here…)*
  - Request pipelining
  - Request multiplexing over single connection


HTTP/2 uses
binary format

HTTP/2 Binary

- Advantages of Binary
  - Simplifies parsing (~~Case? Line endings? Whitespace?~~)
  - Compact representation
  - Simplifies multiplexing and prioritization
  - Impact: Reduces latency, improves throughput
- Disadvantages of Binary
  - Can't fire up Telnet to port 80 and demonstrate HTTP any more
    (couldn't do that with HTTPS anyway…)

HTTP/2 Protocol

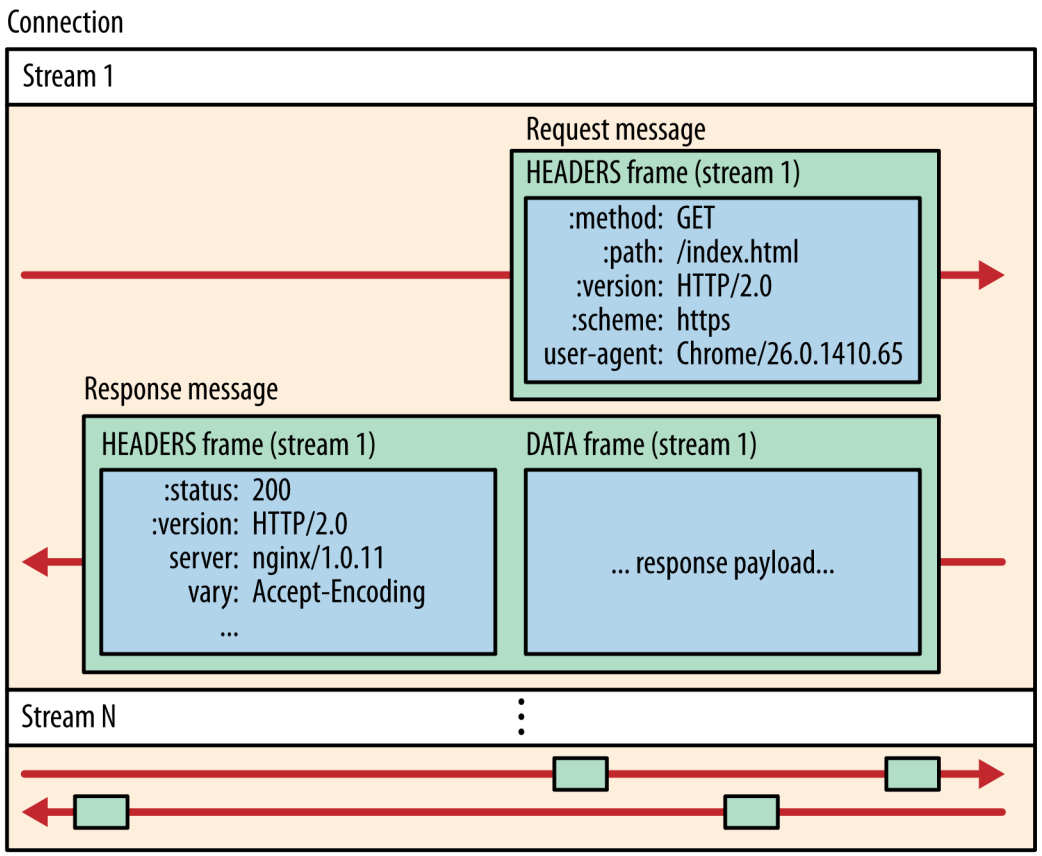


https://developers.google.com/web/fundamentals/performance/http2/

HTTP/2 Protocol

- **Stream** = Bidirectional flow of data
  - Streams can carry 1 or more messages
  - Binary format allows multiple streams to exist over single TCP connection
- **Message** = Collection of frames that, combined, form a *request* or *response* message
- **Frame** = Fundamental unit of communication
  - *Note that these are frames* inside *the TCP connection, not Ethernet frames…*

HTTP/2 Protocol



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

HTTP/2 Protocol

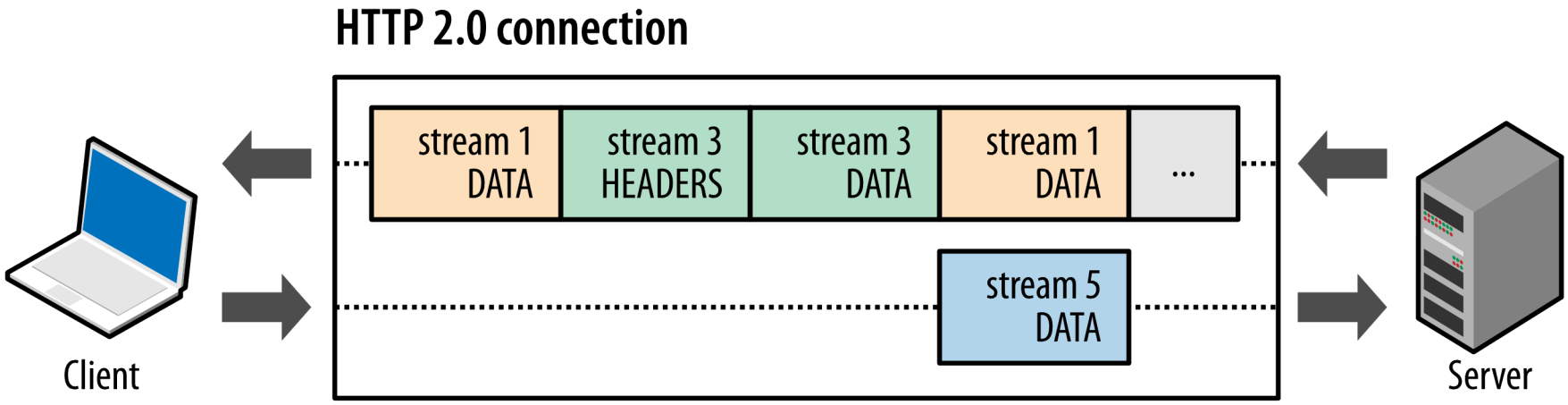
Frame Format

- Length
- Type
- Flags
- Stream Identifier
- *Payload*

Frame Types

- DATA
- HEADERS
- PRIORITY
- PUSH_PROMISE
  - Server push
- WINDOW_UPDATE
  - Flow control
- *(10 frame types in total)*

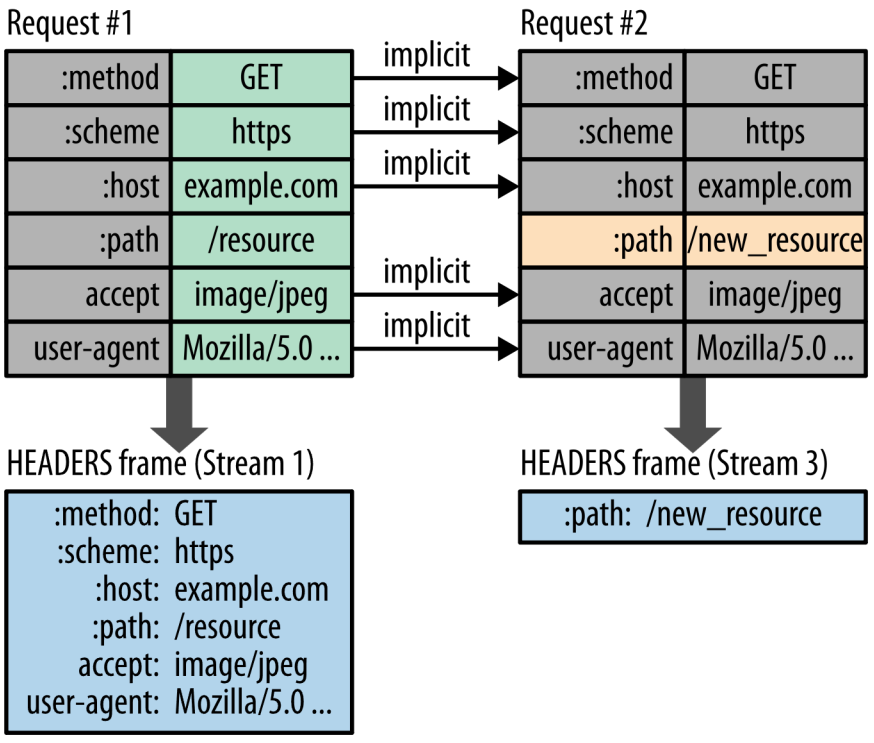
HTTP/2 Multiplexing



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

- Interleave multiple requests / responses without head-of-line blocking
  - Ability for client or server to prioritize frames
- Single TCP connection to each "origin" (host)
  - Connection overhead and initial latency (especially TLS) is amortized over multiple file transfers
  - More efficient than opening multiple connections

HTTP/2 Compression



- **HPACK Header Compression (RFC 7541)**
- Fields can be encoded with static Huffman code
  - Reduces transfer size
- Client and server maintain list of previously-seen fields
  - No need to re-transmit duplicate values (refer to them by index number)
- Operates with bounded memory requirements (i.e. embedded systems)
- Resistant to security flaws (e.g. CRIME – Compression Ratio Info-leak Made Easy)
  - Steal HTTP cookies from TLS connection by observing impact of random payloads on compressed ciphertext length

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

HTTP/2 Compression

- **Is header compression worth the code?**
- Typical page
  - ~75 assets
  - ~1400 bytes per request (referrer tag, cookies, …)
- 7-8 round trips just to transmit requests
- Limited by TCP slow start congestion control
  - Can only have a few outstanding packets until ACKs begin returning
- Benefits increase the greater your network latency
  - Mobile LTE latency: 100+ ms (best case)

HTTP/2 Security

- HTTP/2 standard (RFC 7540) supports unencrypted connections
- Subject of much debate! (Should encryption be mandatory?)
  - Pros:
    - Security security security
    - Prevents tampering from *annoying* middleboxes assuming anything over port 80 is plain HTTP/1.1
  - Cons:
    - Not all content has to be encrypted
    - Performance / latency / etc…

HTTP/2 Security

- HTTP/2 standard (RFC 7540) supports unencrypted connections
  - Client starts with HTTP/1.1 and sends header: `Upgrade: h2c`
  - Server responds with `HTTP 101 Switching Protocol` status code
  - Rare in wild! (*curl?*)
- **Major browser implementations (Firefox, Chrome, Safari, etc…) only support HTTP/2 over TLS connections**
  - Consistent with overall design philosophy of HTTPS everywhere: New features only enabled over HTTPS

HTTP/2 Protocol

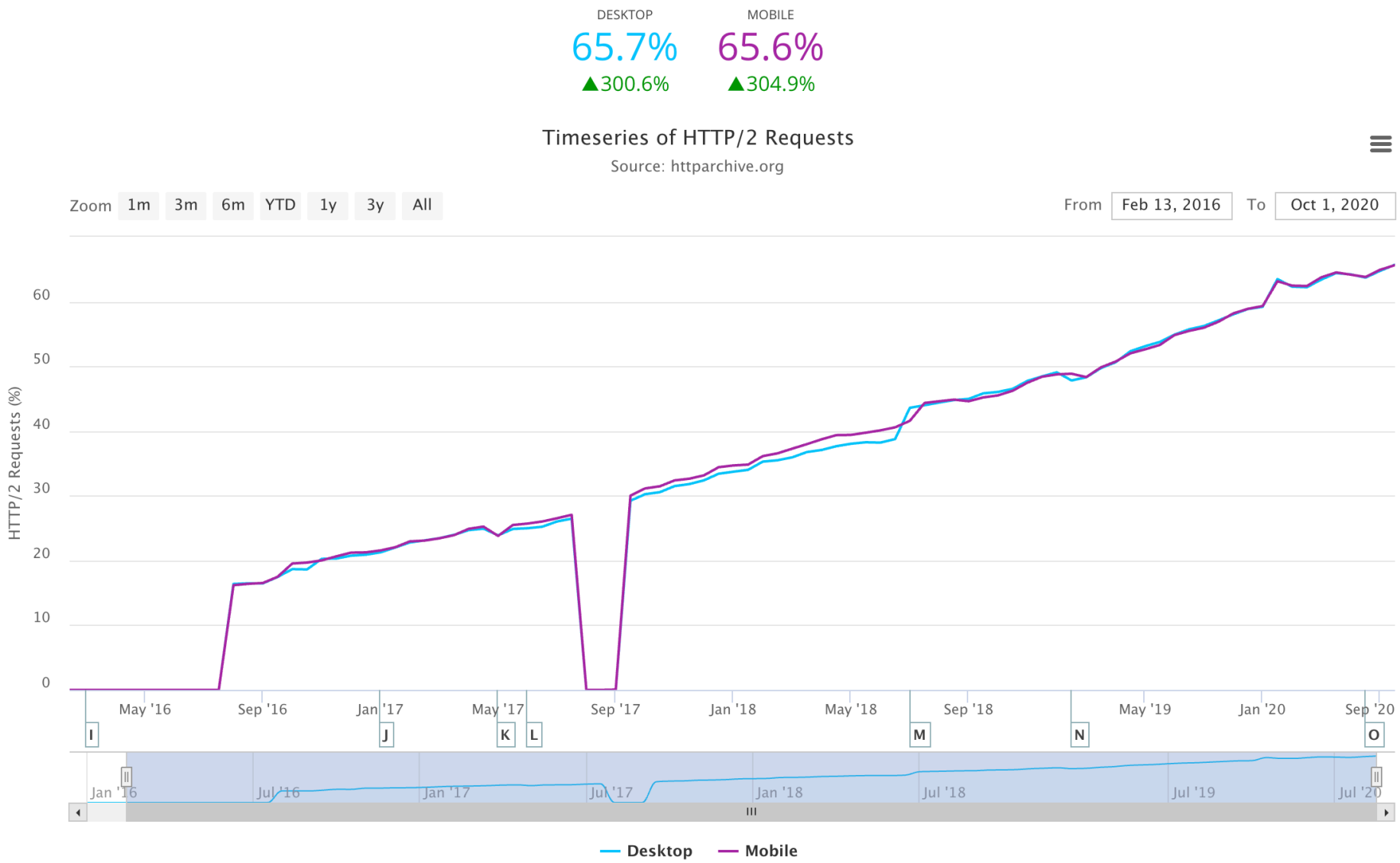
- How to “enable” HTTP/2?
- Application Layer Protocol Negotiation (ALPN) – RFC 7301
  - Part of TLS handshake
  - Client provides server with list of protocols it supports
  - Server picks one it prefers
  - Performance optimization that avoids additional roundtrip of starting with HTTP/1.1 and upgrading to HTTP/2

HTTP/2 Adoption

- **Widespread support!**
- Web browsers
  - Chrome, Safari, Firefox, Edge, …
- Web servers
  - Apache, nginx, IIS, …
- Content delivery networks
  - Akami, Azure, Cloudflare, AWS CloudFront, …

HTTP/2 Requests

The percent of all requests in the crawl using HTTP/2.



https://httparchive.org/reports/state-of-the-web

HTTP/3 (and "Google QUIC")

It’s a Google thing….

(Originally)

Google Engineering Motivations

- **Goal: Decrease end-user latency on web**
  - *To increase user engagement…*
  - *So they see more ads…*
- Approaches
  - Expand Google's content delivery network to be physically closer to audience
    - Fewer network hops, fewer wire delays
  - Develop and optimize web browser (Chrome)
  - Update HTTP protocol (HTTP/2)

Google Engineering Motivations

- **HTTP/2** (based on Google SPDY)
- Decrease end-user latency via
  - Compressing HTTP headers (fewer bits)
  - Pipelining requests
  - Multiplexing many HTTP requests onto single TCP connection
  - ~~Allowing server to push *anticipated* content to users~~

How do you make the web faster?

Google

User-perceived latency

$BROWSER

HTTP/1.1

TLS 1.2

TCP

IP

Physical Network

google.com

Launch your own browser

Update HTTP

Chrome

HTTP/2

What about this part of the stack?

Build a carrier-grade network

Google CDN

google.com

https://www.nanog.org/sites/default/files/meetings/NANOG64/1051/20150603_Rogan_Quic_Next_Generation_v1.pdf

Google Engineering Motivations

- Problems demonstrated in HTTP/2 testing
  - TCP is **in-order delivery** protocol
    - All packets are precious!
    - **Head-of-Line problem:** Loss of a single packet prevents delivery of all behind it until (slow) retransmission occurs
  - If multiple streams are being sent through *single* TCP connection, all are delayed
- Can we do better?
  - Challenge: TCP is baked into the operating system kernel (Windows, OS X, Linux) – Difficult for even Google to modify

Transmission Control Protocol (TCP)

- TCP is connection-oriented
  - 3-way handshake used for connection setup
- TCP provides a stream-of-bytes service
- TCP is reliable
  - Acknowledgements indicate delivery of data
  - Checksums are used to detect corrupted data
  - Sequence numbers detect missing, or mis-sequenced data
  - Corrupted data is retransmitted after a timeout
  - Mis-sequenced data is re-sequenced
  - (Window-based) Flow control prevents over-run of receiver
- TCP uses congestion control to share network capacity among users

QUIC Overview

- **Q**uick **U**DP **I**nternet **C**onnections
  - Roughly, TCP+TLS+HTTP/2 on top of UDP
- Design Goals
  - Provide multiplexed connections between two hosts (without head-of-line blocking)
  - Provide security (equivalent to TLS) – always encrypted
  - Reduce connection establishment latency
  - Improve congestion control
  - Provide bandwidth estimation to applications to avoid congestion
  - "Innovate" at the userspace (not constrained by OS kernel, legacy clients, middleboxes)

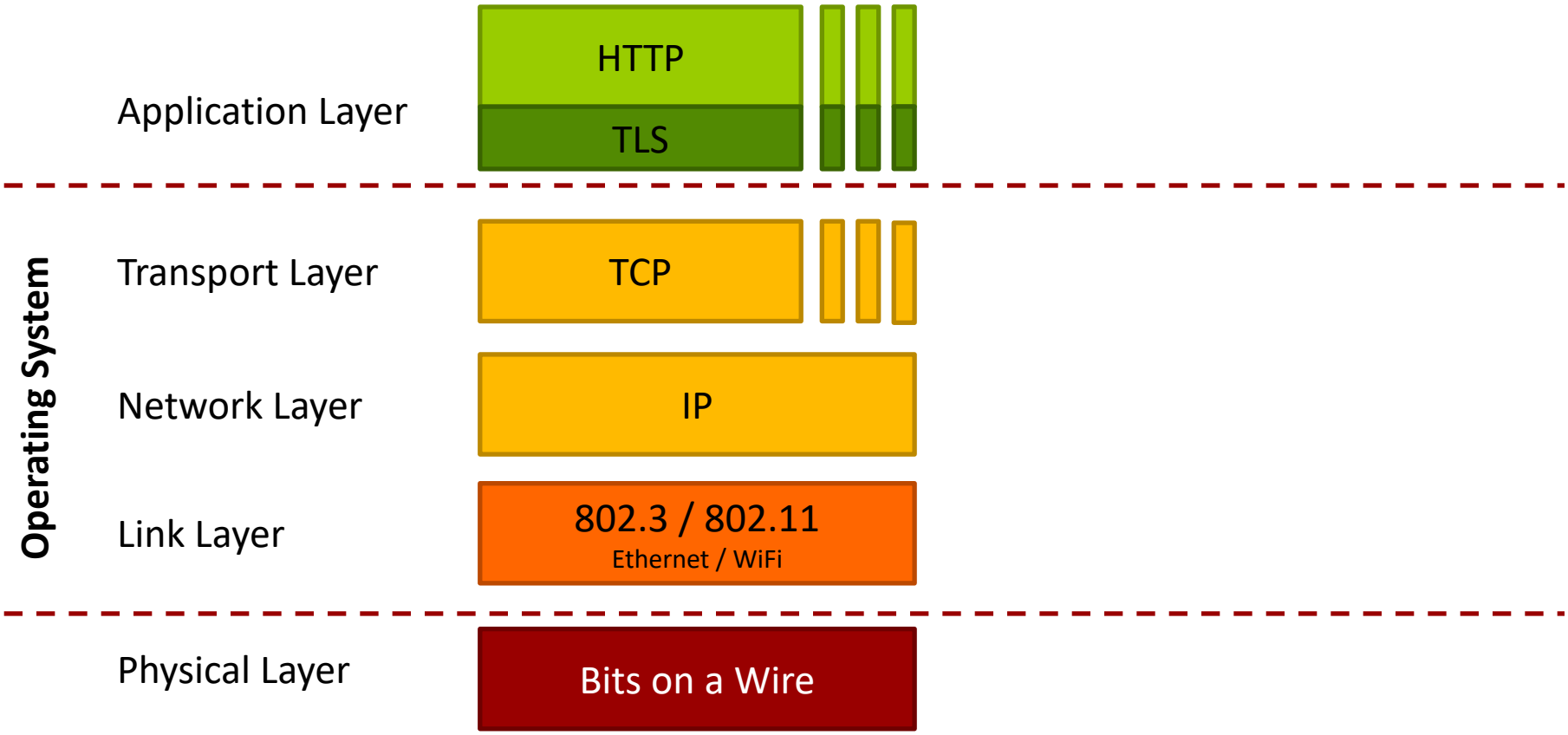
User Datagram Protocol (UDP)

- UDP is a connectionless datagram service
  - There is no connection establishment: packets may show up at any time
- UDP packets are self-contained
- UDP is unreliable
  - No acknowledgements to indicate delivery of data
  - Checksums cover the header, and only optionally cover the data
  - Contains no mechanism to detect missing or mis-sequenced packets
  - No mechanism for automatic retransmission
  - No mechanism for flow control or congestion control (sender can overrun receiver or network)

Traditional Architecture

HTTP/1.1 on top of TCP

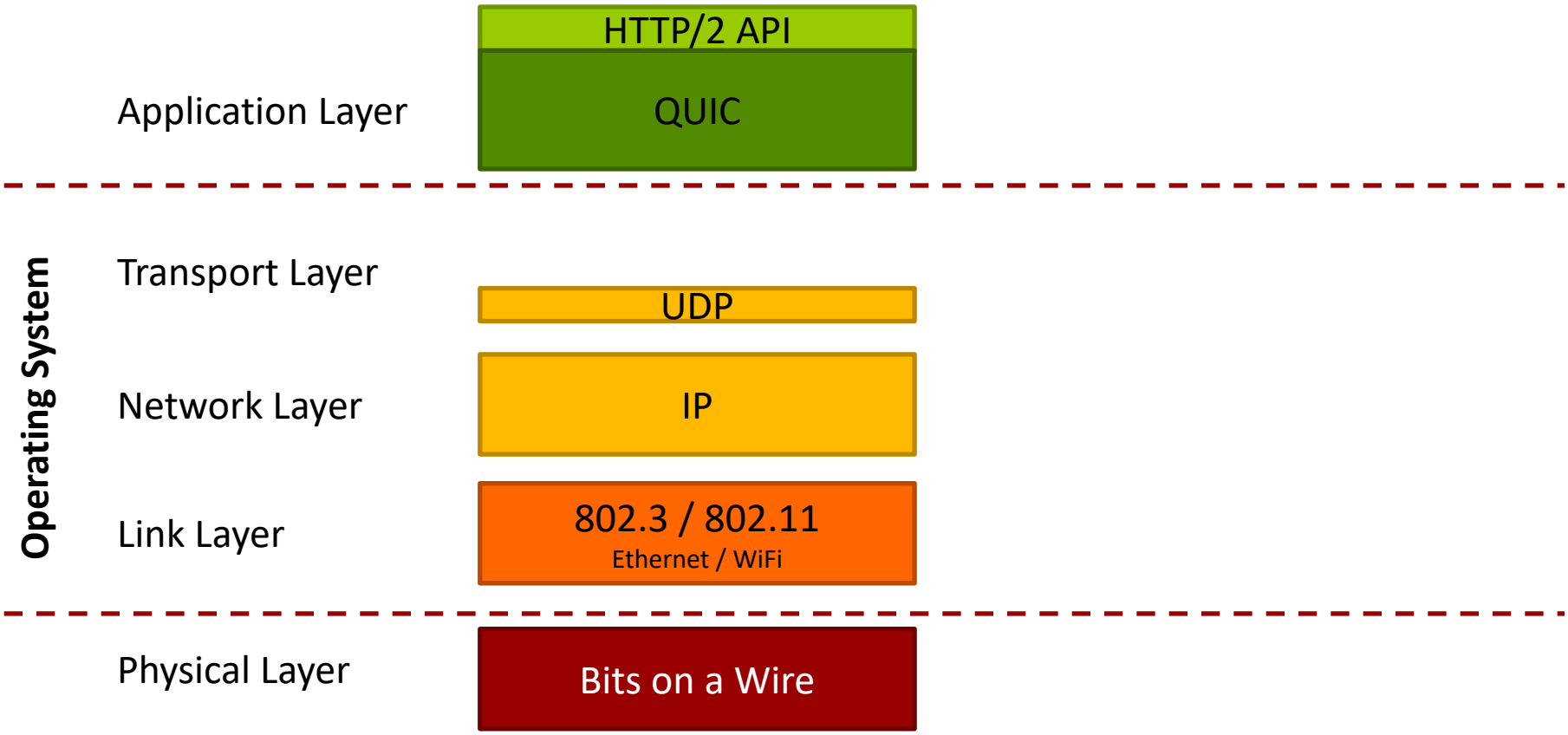
Multiple connections to same server (for parallelism) require multiple TCP sockets



QUIC Architecture

HTTP/2 on top of QUIC

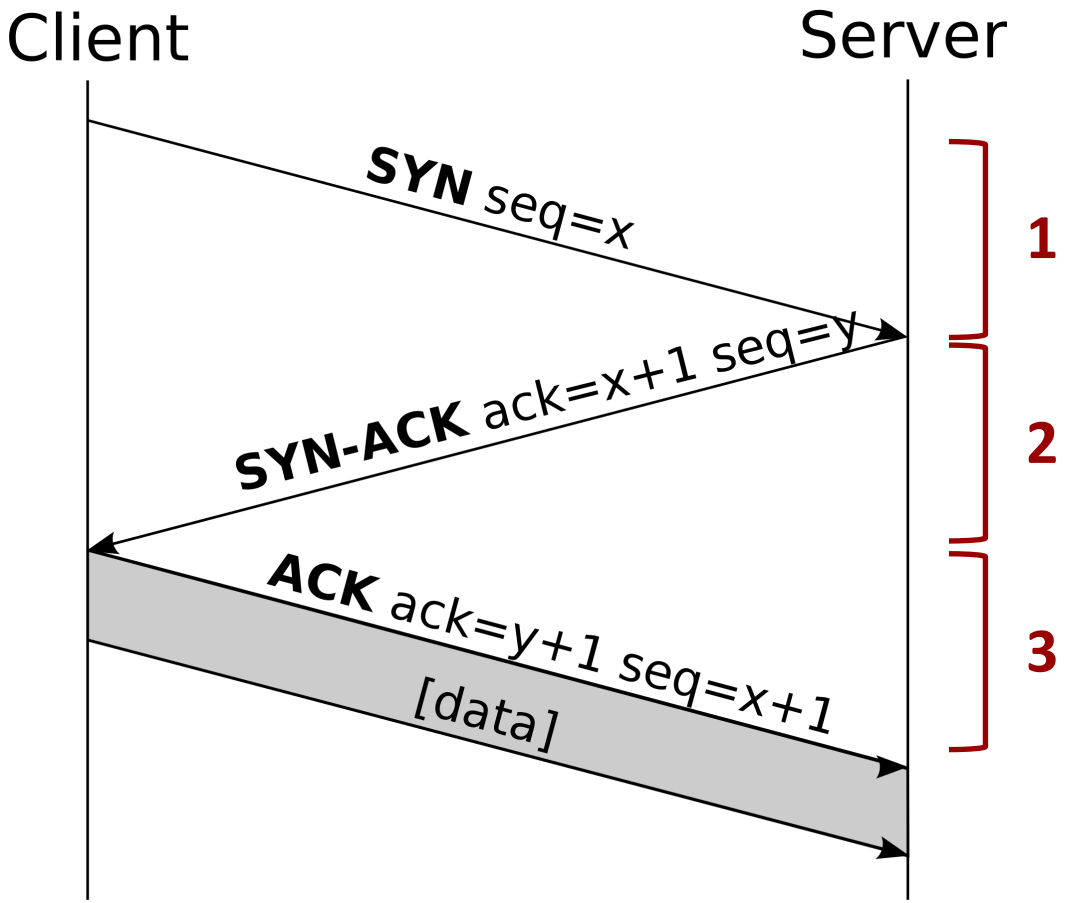
Single UDP stream presented to operating system (multiplexed)



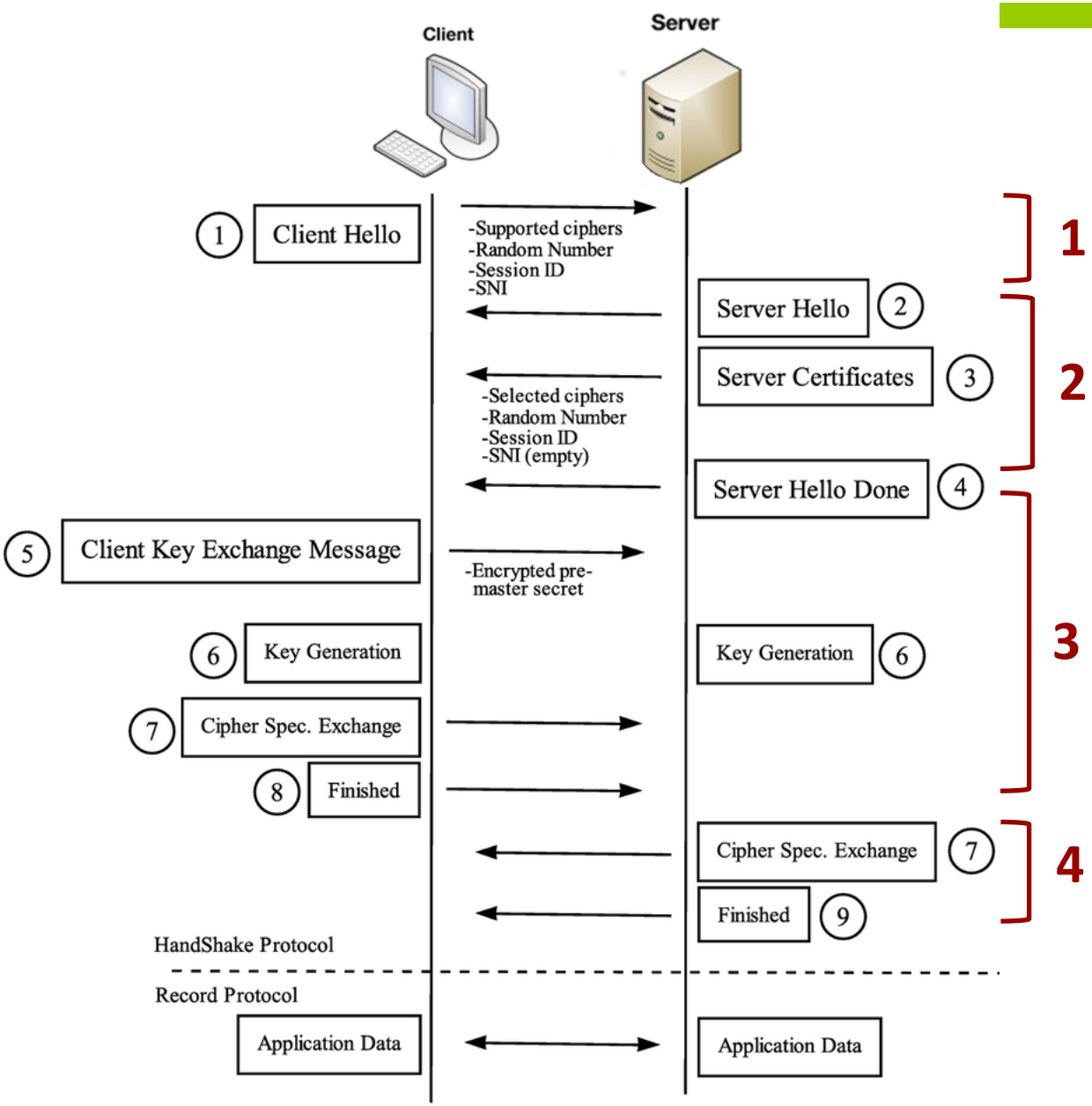
QUIC vs TCP

- QUIC uses server UDP port 443 instead of TCP 443
- QUIC is fully encrypted by default
  - Except for flags, connection ID, and sequence number
  - Avoids network ossification by “helpful” network operators and middleware boxes
- QUIC retransmits data with new sequence numbers and re-encrypts them
  - Improves loss recovery and RTT measurement
- QUIC has no head of line blocking
  - Only the stream with the missing packet is blocked
  - All other streams can continue

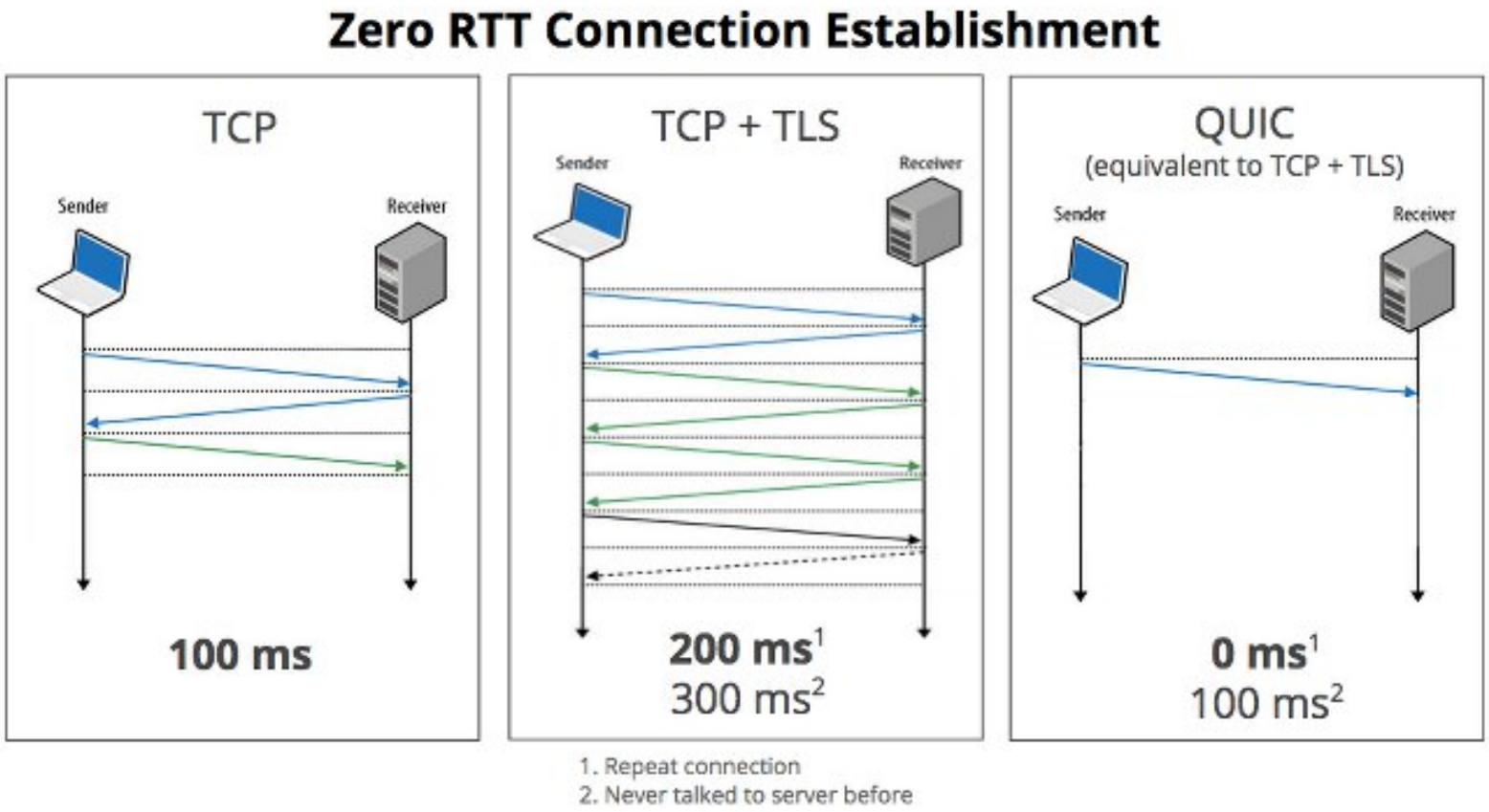
TCP 3-Way Handshake



TLS Handshake



QUIC vs TCP



https://peering.google.com/#/learn-more/quic

QUIC Zero RTT Connections

- QUIC Connection Setup
  - 0 round-trips to a known server (common)
  - 1 round-trip if crypto keys are not new
  - 2 round-trips if QUIC version negotiation needed
  - After setup, HTTP requests/responses flow over connection
- QUIC inspired TLS 1.3 Zero RTT handshake

https://datatracker.ietf.org/meeting/98/materials/slides-98-edu-sessf-quic-tutorial-00.pdf

QUIC Congestion Control

- QUIC builds on decades of experience with TCP
- Incorporates TCP best practices
  - TCP Cubic - fair with TCP FACK, TLP, F-RTO, Early Retransmit...
- Adds signaling improvements that can't be done to TCP
  - Retransmission uses a new sequence number
  - Avoid ambiguity about which packets have been received

Mobile

- QUIC better supports mobile clients
  - Handing off between WiFi and cell network
  - Switching apparent IP addresses
- QUIC token allows a client to continue with an established connection even if the IP address changes

Performance

- Strong network connection?
  - 3% latency improvement
- Weak network connection?
  (99% percentile of connections to Google search)
  - Reduced page loading time by **1 second**
  - Strong benefit over TCP on marginal internet connections (third world/emerging markets, high latency satellite Internet, lousy mobile devices over weak WiFi, etc…)
- YouTube
  - 30% fewer rebuffers (video pauses)

Standardization as HTTP/3

- Internet Engineering Task Force (IETF)
  Working Group formed in 2016 - https://quicwg.org/
  - **Draft HTTP/3 standard published November 21st, 2020**
- Not just adopting Google QUIC
  - Google's QUIC was an experiment, tested on a large scale, yielding valuable data
  - Replacing "Google crypto" with TLS 1.3
  - Standardizing APIs
  - New packet format (long headers and short headers)

HTTP/3 Adoption

- **Under Development**
- Web browsers
  - Chrome (79+), Safari (14+), Firefox (72+), Edge (Canary build+)
  - Typically *disabled by default*
- Web servers & content delivery networks
  - "Experimental / technology preview stage" (CloudFlare is production…)
- **Enabled at Facebook**
  - "More than 75 percent of our internet traffic uses QUIC and HTTP/3"
    - Facebook App and Instagram App, not website
  - https://engineering.fb.com/2020/10/21/networking-traffic/how-facebook-is-bringing-quic-to-billions/
- **Enabled at Google**