CS 188

Scalable Internet Services

John Rothfels
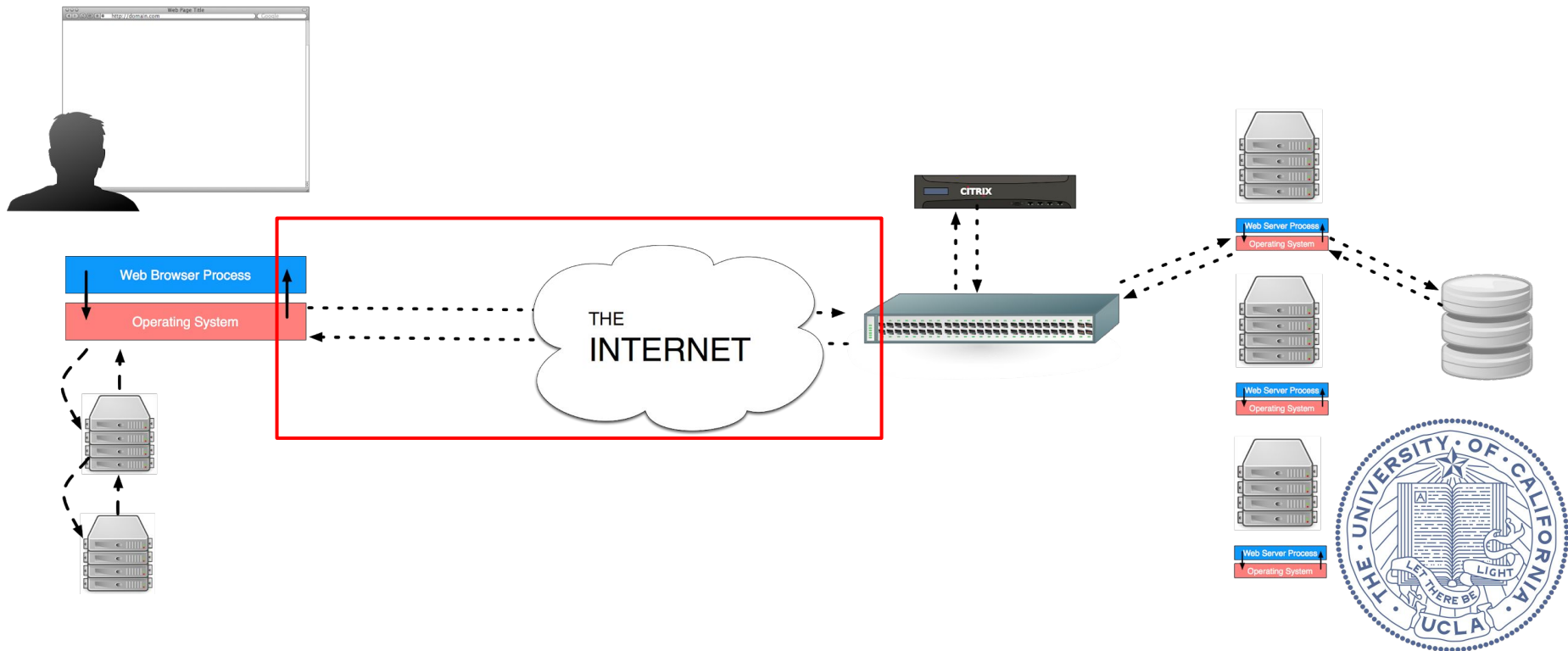Nov 19, 2019

Agenda

- HTTP/2
- QUIC
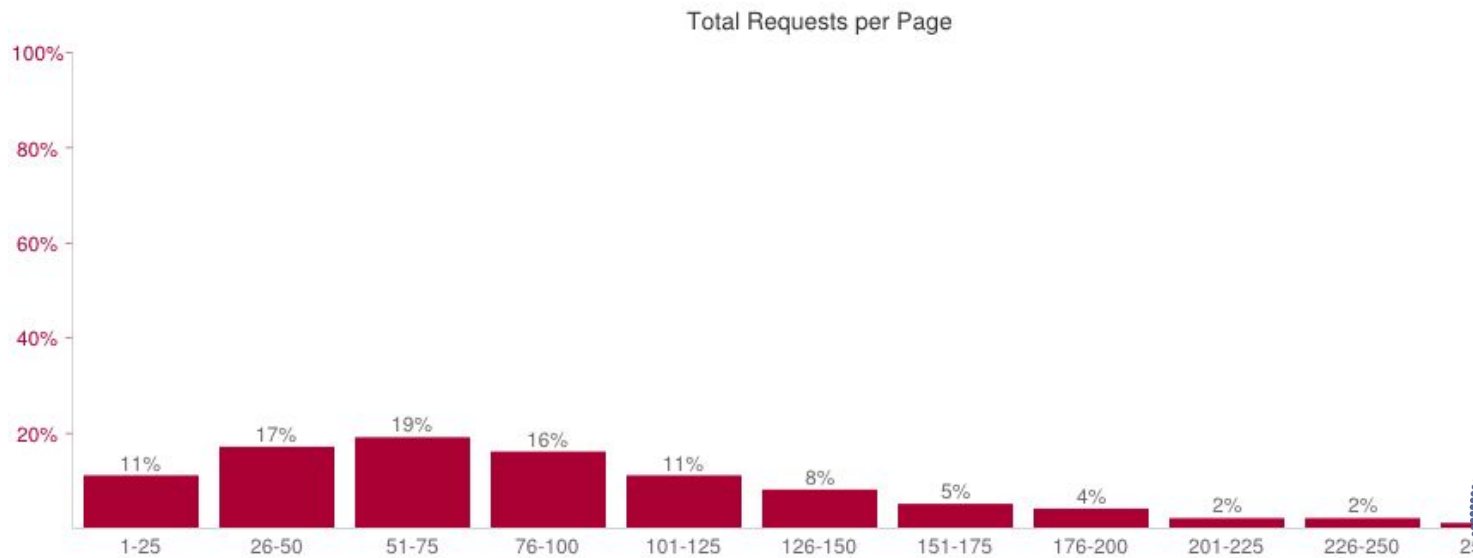
Motivation

Today's Performance Problems

Web pages have many constituent resources

Total Requests per Page

- 1-25: 11%
- 26-50: 17%
- 51-75: 19%
- 76-100: 16%
- 101-125: 11%
- 126-150: 8%
- 151-175: 5%
- 176-200: 4%
- 201-225: 2%
- 226-250: 2%

Source: http://httparchive.org/

Today's Performance Problems

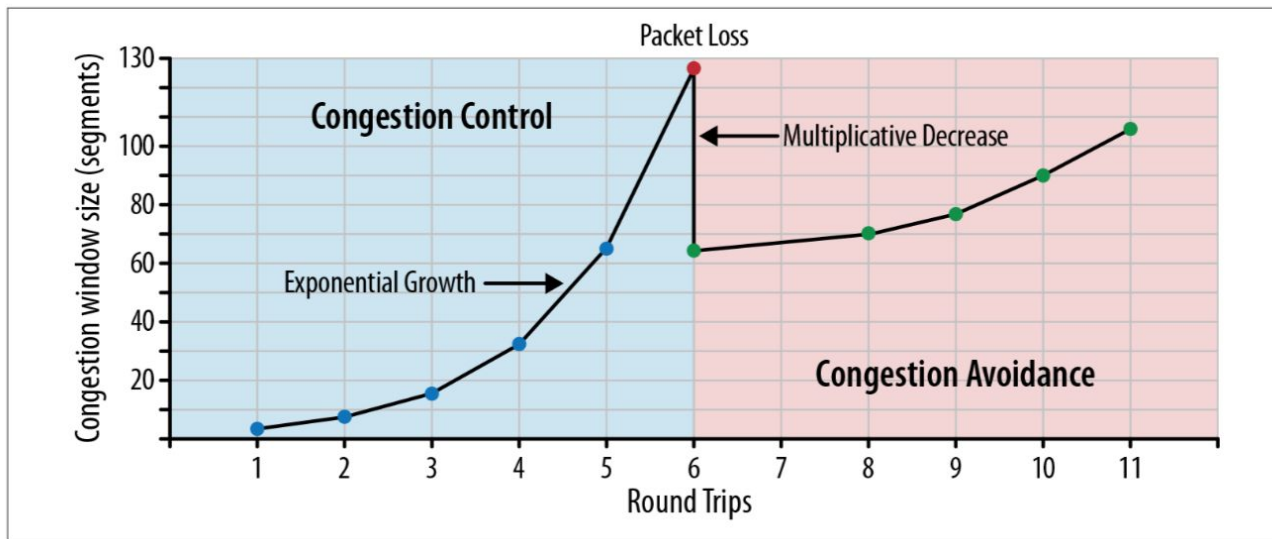Many requests are needed to present today's web pages.

- CSS
- Javascript
- Images

Establishing many TCP connections to serve all these is very slow. Why?
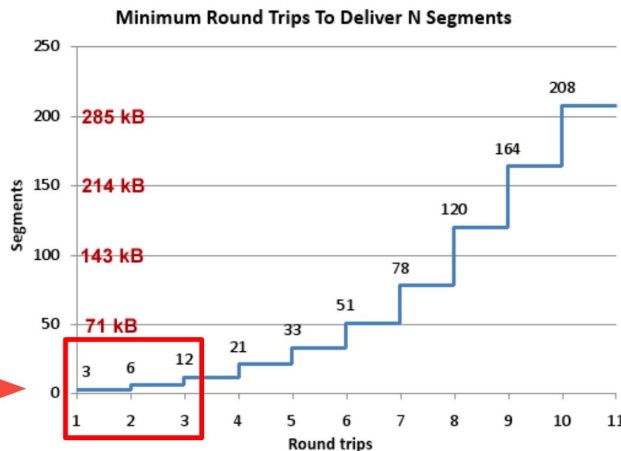
Today's Performance Problems

Establishing many TCP connections to serve all these is very slow.

Today's Performance Problems
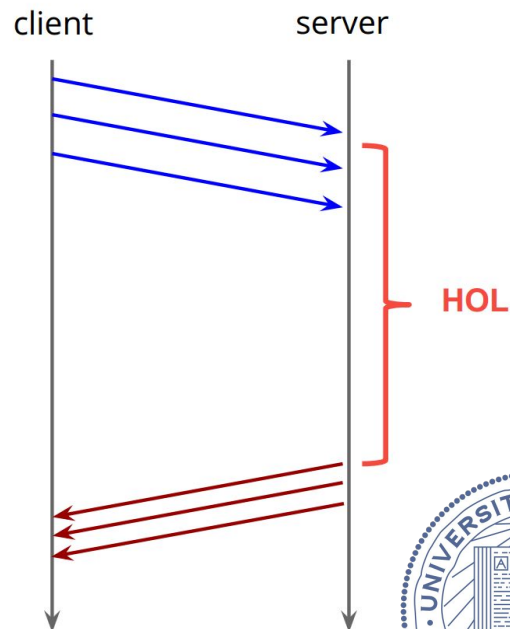
TCP was designed for long-lived flows. HTTP is short and bursty.

Today's Performance Problems

HTTP Keepalive was introduced to help (and it does), but there are problems.

- We can reuse a TCP socket for multiple HTTP requests, but one heavyweight request can affect all others
- This is called Head-of-line blocking

Today's Performance Problems

Additionally, if you look at the data that is being sent, there is a lot of repetition.

```
GET / HTTP/1.1
Host: www.etsy.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_2) AppleWebKit/536.26.14 (
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
DNT: 1
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Cookie: uaid=uaid%3DVdhk5W6sexG-_Y7ZBeQFa3cq7yMQ%26_now%3D1325204464%26_slt%3Ds_L(
Connection: keep-alive
```

525 bytes

Today's Performance Problems

Additionally, if you look at the data that is being sent, there is a lot of repetition.

```
GET /assets/dist/js/etsy.recent-searches.20121001205006.js HTTP/1.1
Host: www.etsy.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_2) AppleWebKit/536.26.14 (
Accept: */*
DNT: 1
Referer: http://www.etsy.com/
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Cookie: autosuggest_split=1; etala=111461200.1476767743.1349274889.1349274889.1349
Connection: keep-alive
```

226 new bytes; 690 total

Today's Performance Problems

Additionally, if you look at the data that is being sent, there is a lot of repetition.

```
GET /assets/dist/js/jquery.appear.20121001205006.js HTTP/1.1
Host: www.etsy.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_2) AppleWebKit/536.26.14 (
Accept: */*
DNT: 1
Referer: http://www.etsy.com/
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Cookie: autosuggest_split=1; etala=111461200.1476767743.1349274889.1349274889.1349
Connection: keep-alive
```

14 new bytes; 683 total

Today's Performance Tricks

Many techniques are used to reduce the number of requests.

- **Why does the asset pipeline exist in Rails?**

Today's Performance Tricks

Many techniques are used to reduce the number of requests.

- Why does the asset pipeline exist in Rails?
  - File concatenation.
  - Having many JS, CSS files means having many requests, so we mash them together.

Today's Performance Tricks

Many techniques are used to reduce the number of requests.

- Image Spriting: the process of putting lots of smaller images in a single image, and then referring to them all using offsets.

Today's Performance Tricks

Today's Performance Tricks

Many techniques are used to reduce the number of requests.

- Image Spriting: the process of putting lots of smaller images in a single image, and then referring to them all using offsets.
  - This spriting is cumbersome to deal with, and is really a hack.

Today's Performance Tricks

Techniques are used to increase the number of parallel requests that browsers can have to a server.

- Most browsers will only open 6 concurrent TCP connections to a single host
  - Why?

Today's Performance Tricks

This is the result:

Today's Performance Tricks

How do we address this?

- We want fewer TCP connections, but...
  - we don't want head-of-line blocking
  - we don't want to have to jam all our css, js artificially together
  - we don't want to have to stuff our images in one big file and deal with offsets everywhere
  - we don't want to have to do DNS tricks to fool the browser

HTTP/2

We address this with HTTP/2

- Started life at Google as SPDY
  - Added to Chrome in 2009
- Pushed towards standardization starting in 2012
- Today supported in all major browsers
- Server side support optional in Apache and Nginx

Standard completed in 2015.

HTTP/2

How does HTTP/2 work?

- One TCP connection, multiplex everything over that
- Header compression
- Server push
- Prioritization

HTTP/2

Single TCP stream: Binary Framed connection
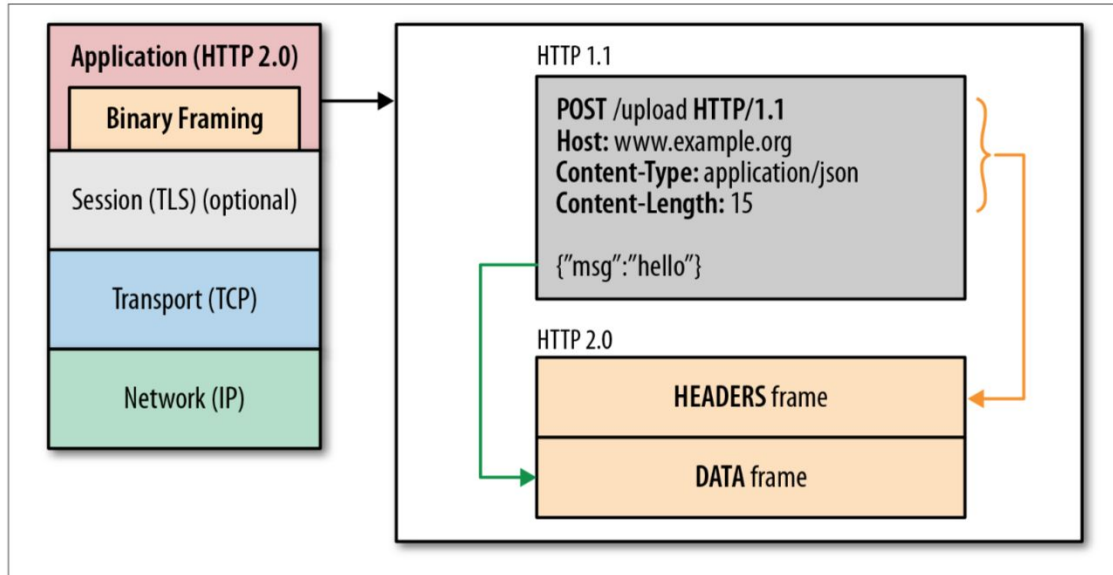
HTTP/2

Single TCP stream: Binary Framed connection
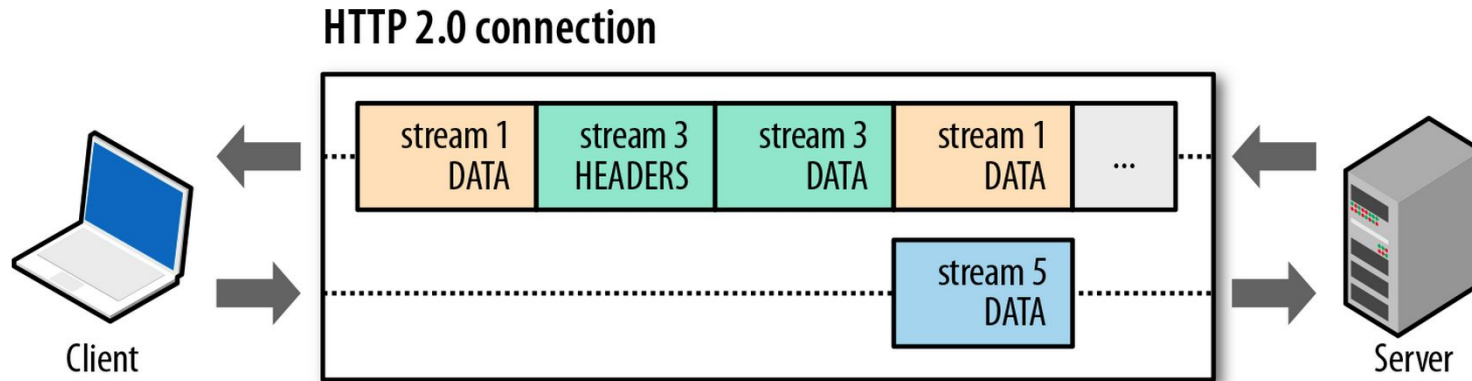
HTTP/2

Single TCP stream: Binary Framed connection
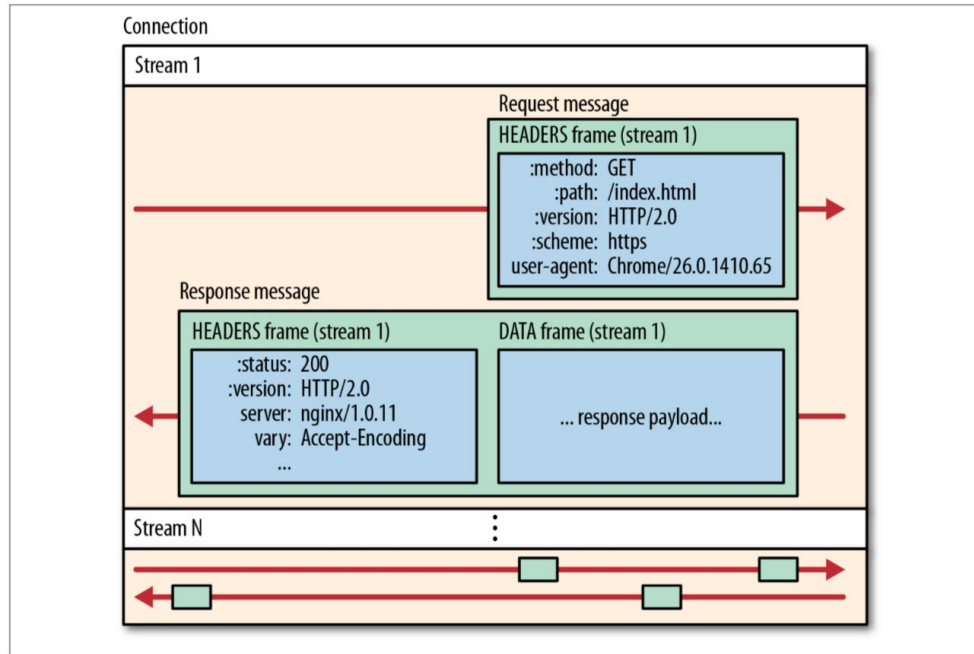
HTTP/2

With binary framing, header compression becomes easy.

- Initially implemented with GZIP, but CRIME attack revealed weaknesses
  - If an attacker can inject data, compressed size can reveal information
- Now uses HPACK

HTTP/2

Request headers

:method	GET
:scheme	https
:host	example.com
:path	/resource
user-agent	Mozilla/5.0 …
custom-hdr	some-value

Static table

1	:authority	
2	:method	GET
…	…	…
51	referer	
…	…	…
62	user-agent	Mozilla/5.0 …
63	:host	example.com
…	…	…

Dynamic table

Encoded headers

2	
7	
63	
19	Huffman("/resource")
62	
Huffman("custom-hdr")	
Huffman("some-value")	
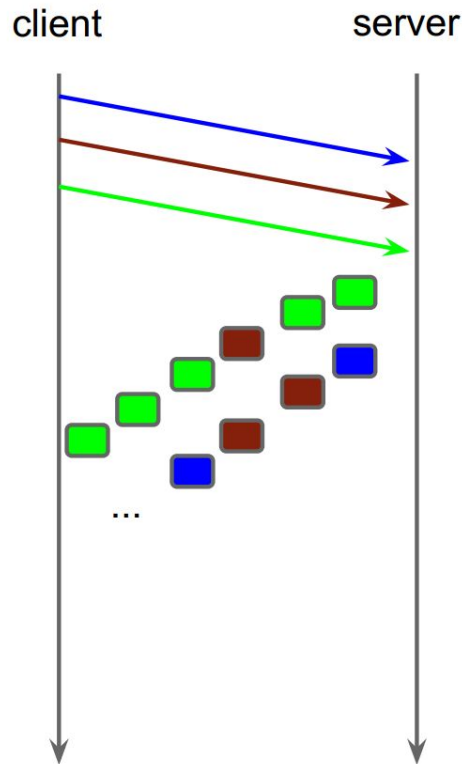
HTTP/2

Binary framing means ordering of resources is flexible.

- Handling of many small resources is efficient
- Headers are compressed, so they are lightweight
- Head of line blocking no longer exists
- No TCP setup burden

HTTP/2

Prioritization & Flow Control

- Since order is no longer mandated, we can implement prioritization
    - DOM highest priority, followed by CSS, Javascript
    - Images lowest

WINDOW_UPDATE flag exists to control number of frames "in flight"

HTTP/2

The hacks of the last 20 years can be thrown out:

- Spriting & asset compliation not needed since we can handle many small resources
- Domain sharding irrelevant since all content is delivered over a single connection

HTTP/2

Server push is now possible
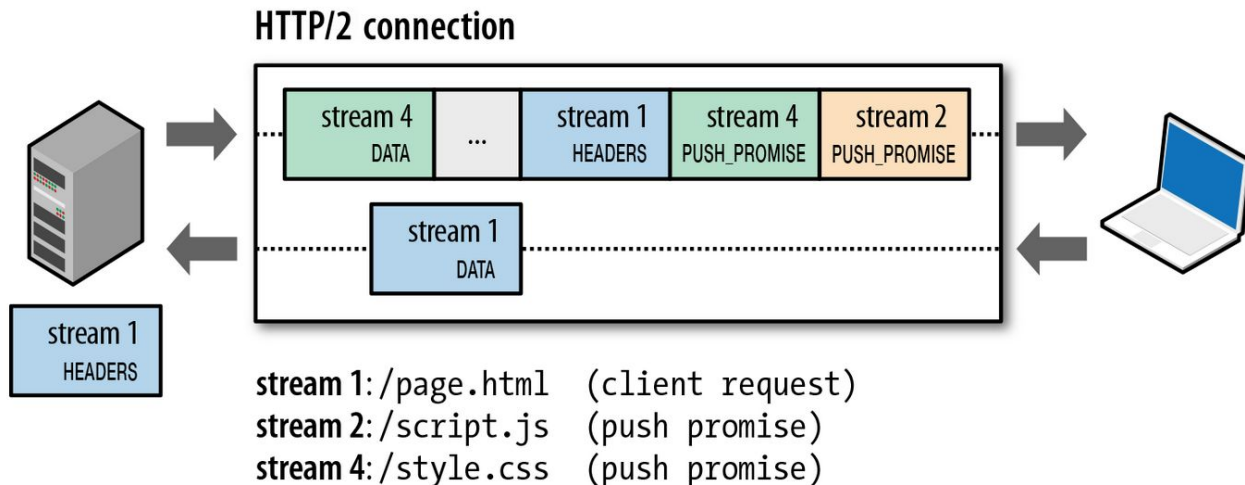
- When a resource is requested, the server can proactively send additional resources using PUSH_PROMISE
- Client can indicate it doesn't want the additional content (e.g. it's cached)

HTTP/2 connection

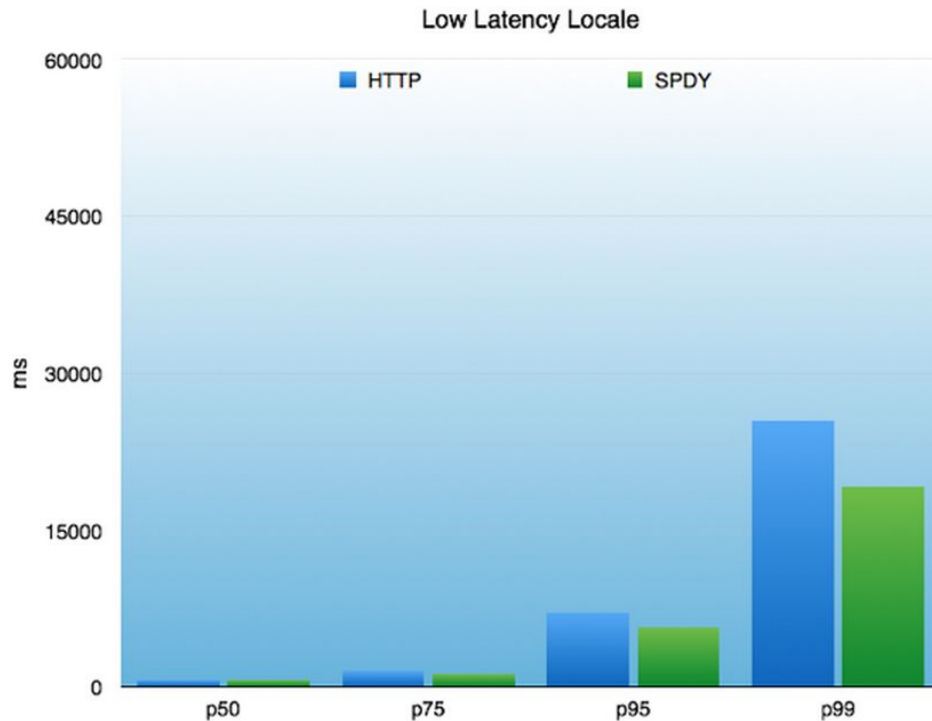| stream 4 DATA | ... | stream 1 HEADERS | stream 4 PUSH_PROMISE | stream 2 PUSH_PROMISE |

stream 1 DATA

stream 1 HEADERS

stream 1: /page.html   (client request)
stream 2: /script.js   (push promise)
stream 4: /style.css   (push promise)

HTTP/2

Results: much faster!

	Google News	Google Sites	Google Drive	Google Maps
Median	-43%	-27%	-23%	-24%
5th percentile (fast connections)	-32%	-30%	-15%	-20%
95th percentile (slow connections)	-44%	-33%	-36%	-28%

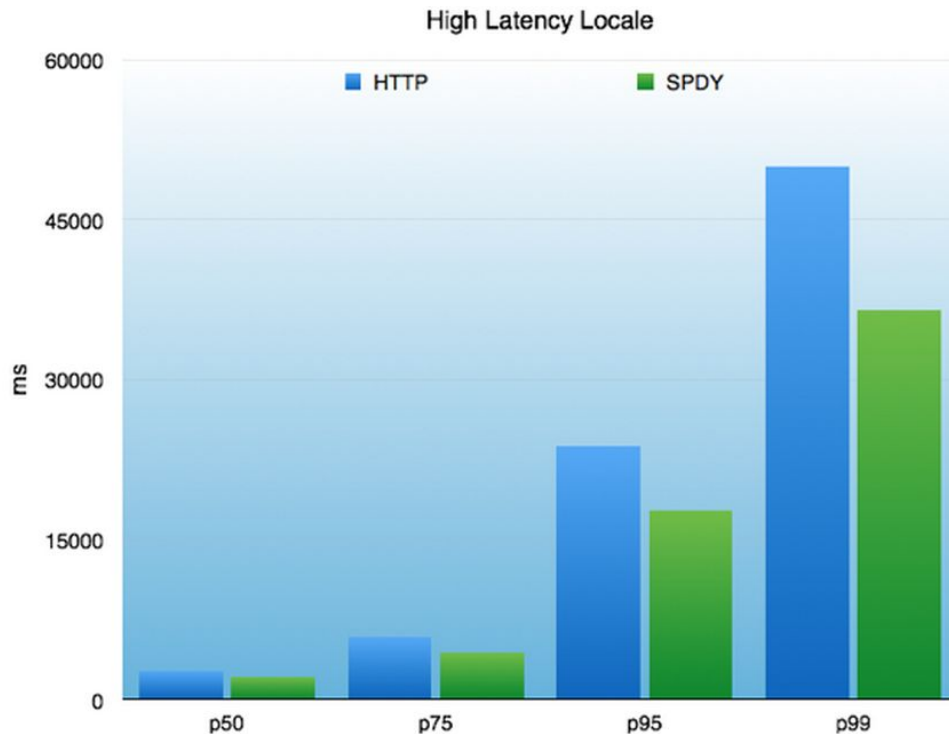time from first request byte to onload event in the browser

HTTP/2

Results:
much faster!

HTTP/2

Results:
much faster!

High Latency Locale

HTTP/2

Demo time:

https://http2.akamai.com/demo

http://www.http2demo.io/

QUIC

HTTP/2 improves HTTP while still using TCP. **QUIC** is an attempt to sidestep TCP in order to sidestep these limitations.
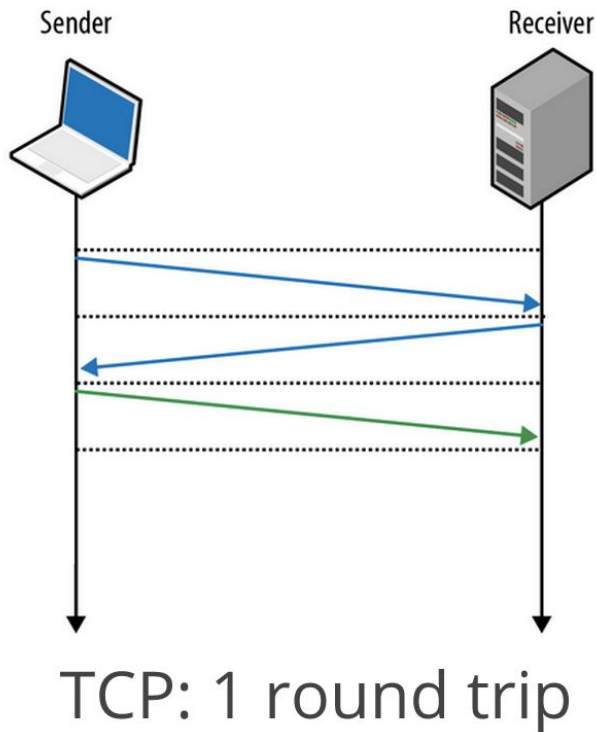- Heavy emphasis on reducing round trips
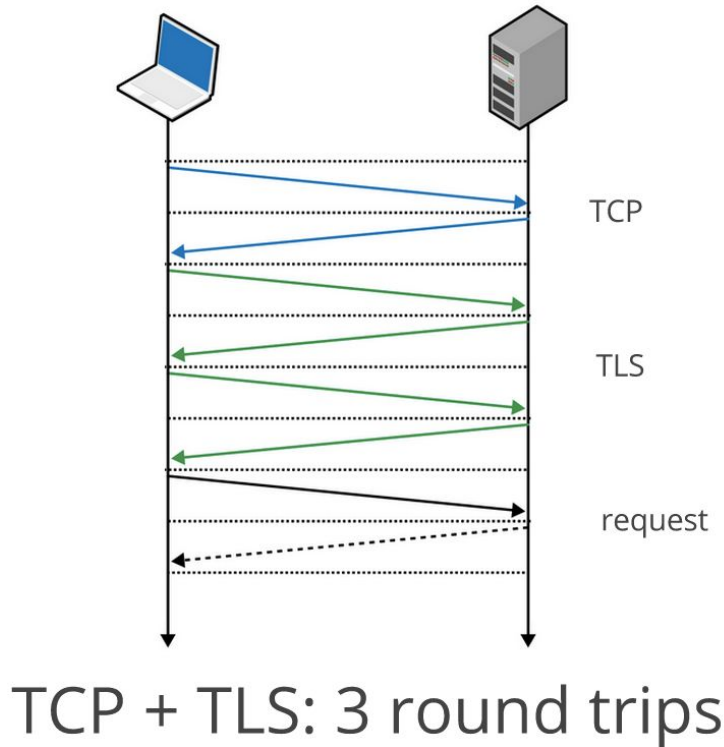
QUIC - Limitations of TCP

TCP: 1 round trip

Connection Establishment:

- TCP connections require a full round trip before any application data can be sent

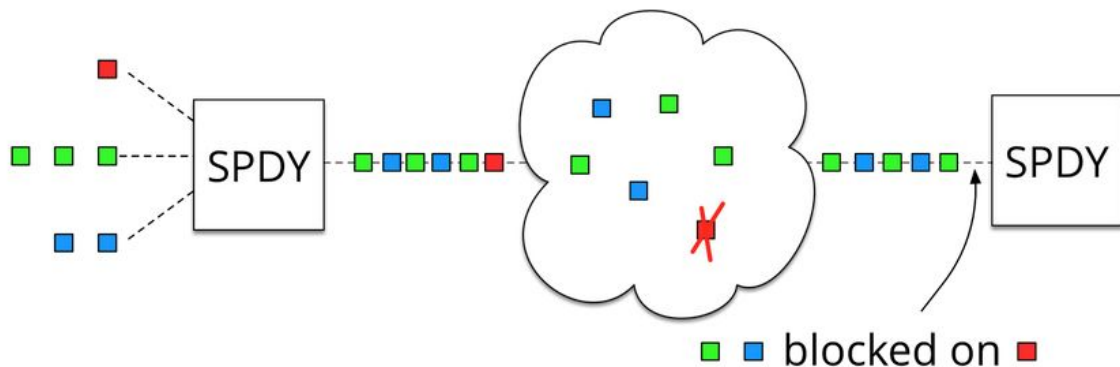# QUIC - Limitations of TCP

TCP + TLS: 3 round trips

Connection Establishment:

- HTTPS requires three full round trips before any application data can be sent
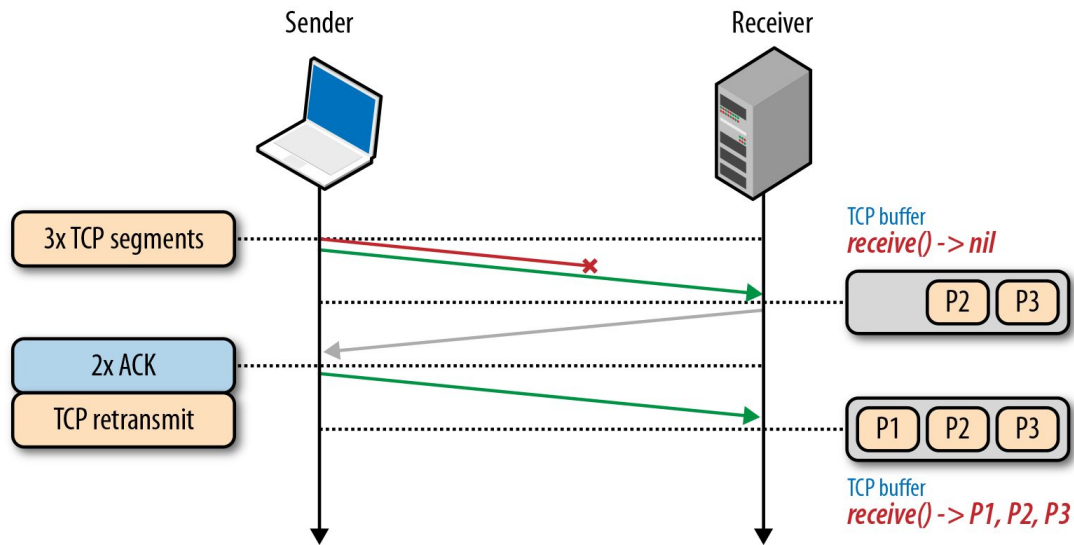
QUIC - Limitations of TCP

Multiplexing:

- We have many independent HTTP/2 requests on a single TCP connection.
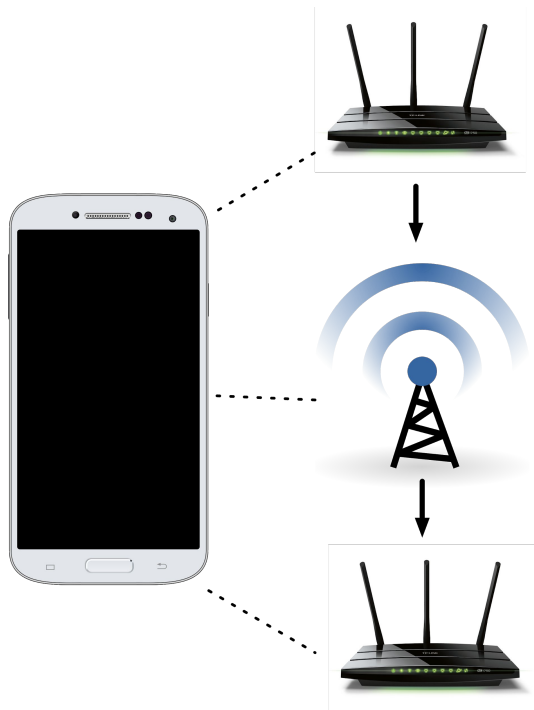- Losing one packet can block unrelated requests

QUIC - Limitations of TCP

Retransmissions:

- Losing packets means retransmissions.
- In high latency networks, this is slow

QUIC - Limitations of TCP

Mobile users:

- Mobile networks have higher latency (100-400ms additional RTT)
- A TCP connection is defined by its source address/port combination.
- If I've got a TCP connection with a good window size open, and I move from wifi to cell, I need to reconnect.

QUIC

Given the limitations of TCP, can we build a better HTTP/2 on top of UDP instead?

- QUIC is an attempt to do this.
- Primary goal is to reduce latency
- You are currently using it if you are using Chrome to talk to Google servers.

QUIC

Initial connection

- Client sends random 64-bit CID
- Server replies with certificate and cookie
- Client responds with CID, cookie, proposed encrypted session key, algorithm
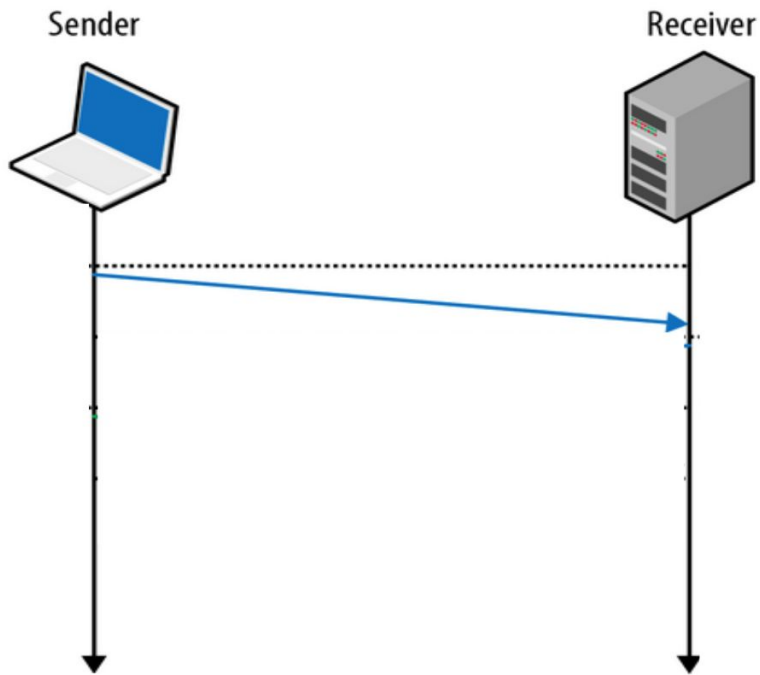- Client can immediately send requests.

QUIC

Subsequent requests

- Client sends CID, cookie, proposed encrypted session key, algorithm
- Client can immediately send requests.
- Assumes server cert hasn't changed

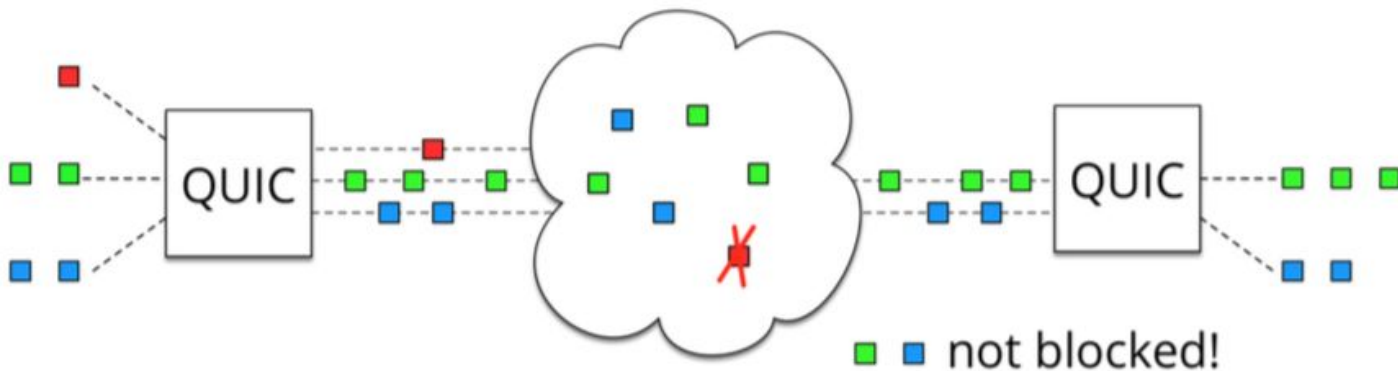Engineered to avoid DOS traffic amplification attacks (IP spoofing)

QUIC

For the common case, initial roundtrips can be completely avoided.

QUIC

Packet loss affects only the resource lost
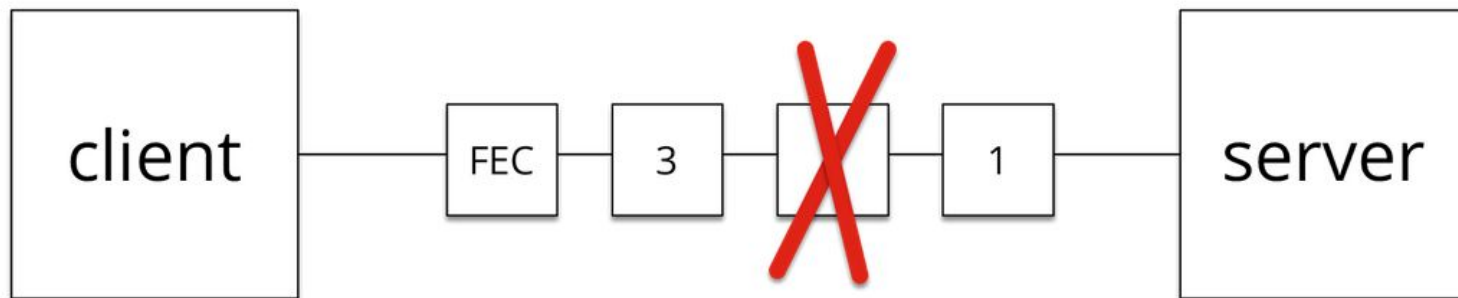
- No head-of-line blocking

QUIC

Traditional TCP handles packet loss through retransmission.

Quic can handle packet loss without retransmission. How?
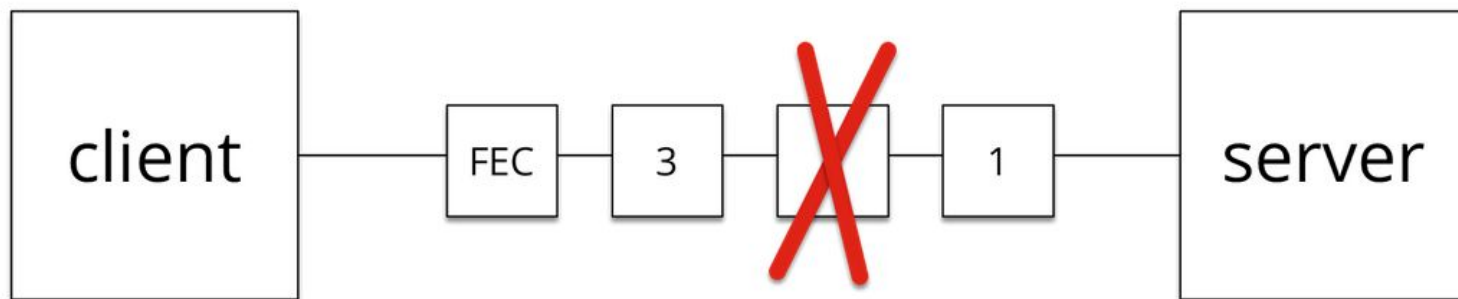
QUIC

Forward Error Correction

QUIC

Note: we are trading bandwidth for latency.
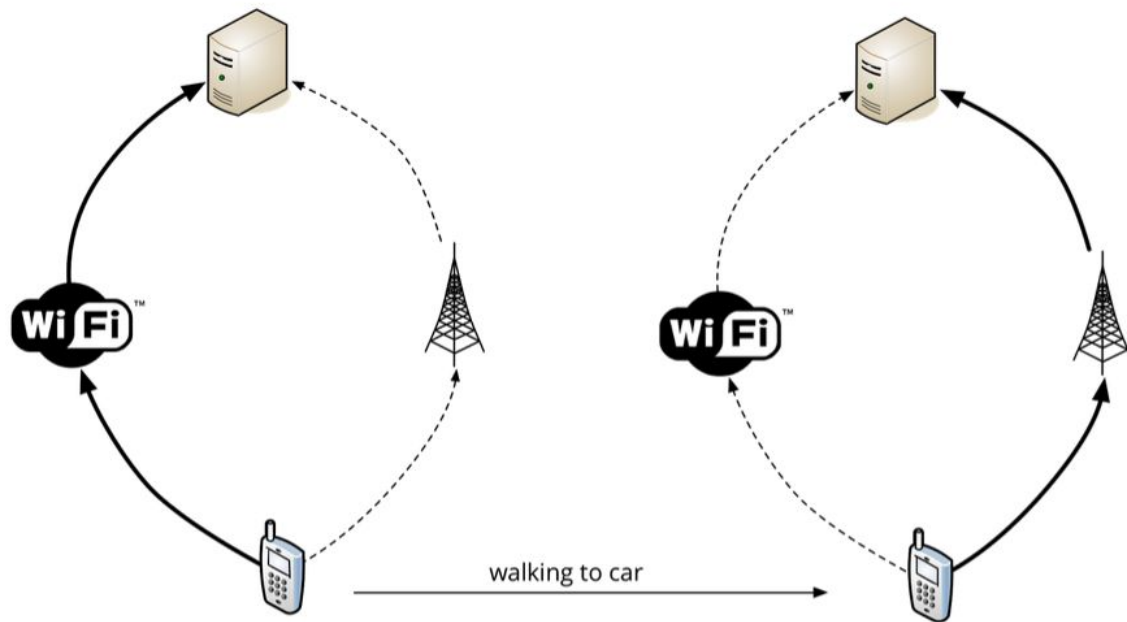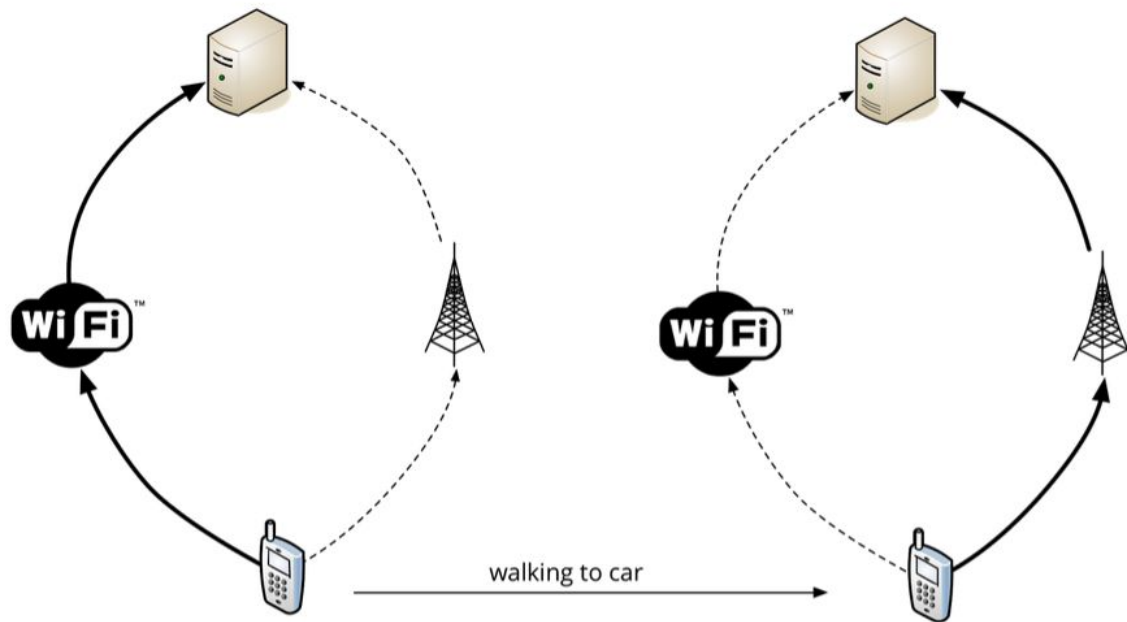
QUIC

Parking lot problem

- Web doesn't handle changing IPs very well
- Mobile users are increasingly common

QUIC

Parking lot problem

- QUIC doesn't expect IP to stay static
- Connection defined by CID

QUIC

Results:

- 75% of requests avoid handshake
- Google search saw 3% reduction in mean page load time.
- For slowest 1% of users, can reduce page load time by a full second.
- YouTube users report 30% fewer rebuffers.
- Performs best under poor network conditions