QUIC for everyone

Expanding QUIC's use in the client OS

Tommy Pauly, EPIQ 2021

Who can use QUIC?

Web browsers

Web browsers

Apps with customized networking libraries

Web browsers

Games

Productivity

OS

VPNs

Video calling

News

Social media

Media streaming

Let’s make it usable for *everyone*

Putting QUIC in the OS

What does QUIC need to be as ubiquitous as TLS/TCP?

- Server support
- Built-in APIs for clients
- Efficiency
- Automatic failover

API support

QUIC APIs

System networking APIs

Available for HTTP/3 as well as “raw” QUIC

Modes of usage

- Raw stream connections, work like TLS/TCP
- Groups of streams for multiplexing awareness

QUIC Streams

TLS stream API

```
let connection = NWConnection(host: "example.com",
                              port: 443,
                              using: .tls)
```

QUIC stream API

```
let connection = NWConnection(host: "example.com",
                              port: 443,
                              using: .quic(alpn: ["h3"]))
```

Sending, receiving, and metrics remain the same

```
connection.send(...
```

Stream Groups

```
let descriptor = NWMultiplexGroup(to: .hostPort(host: "example.com",
                                                port: 443))
let group = NWConnectionGroup(with: descriptor,
                              using: .quic(alpn: ["myproto"]))
```

Outbound stream creation

```
let connection = NWConnection(from: group)
```

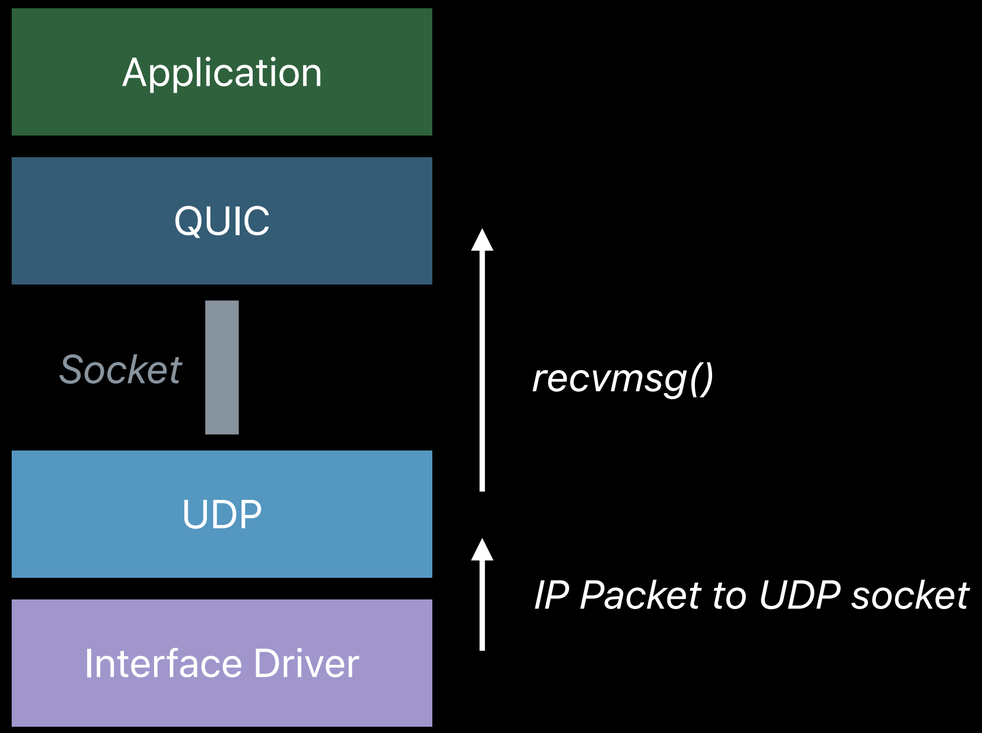
Inbound stream creation

```
group.newConnectionHandler = { newConnection in ... }
```

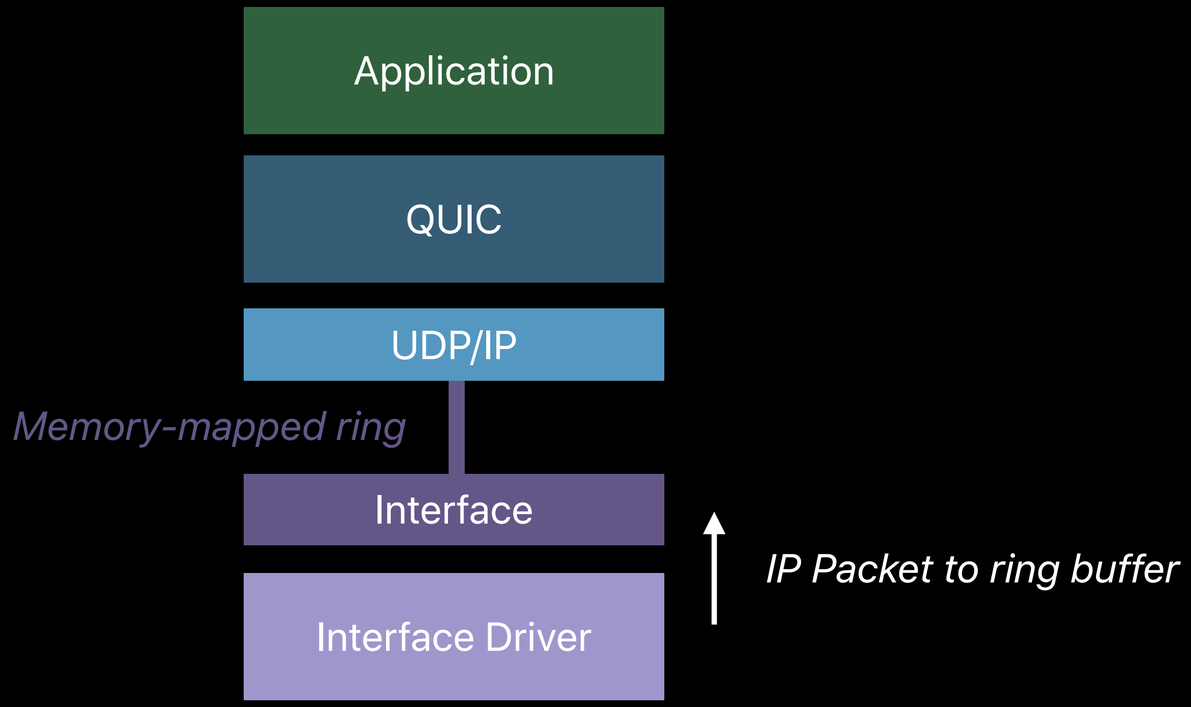
Efficiency

Efficiency
=
reducing UDP socket overhead

UDP Sockets

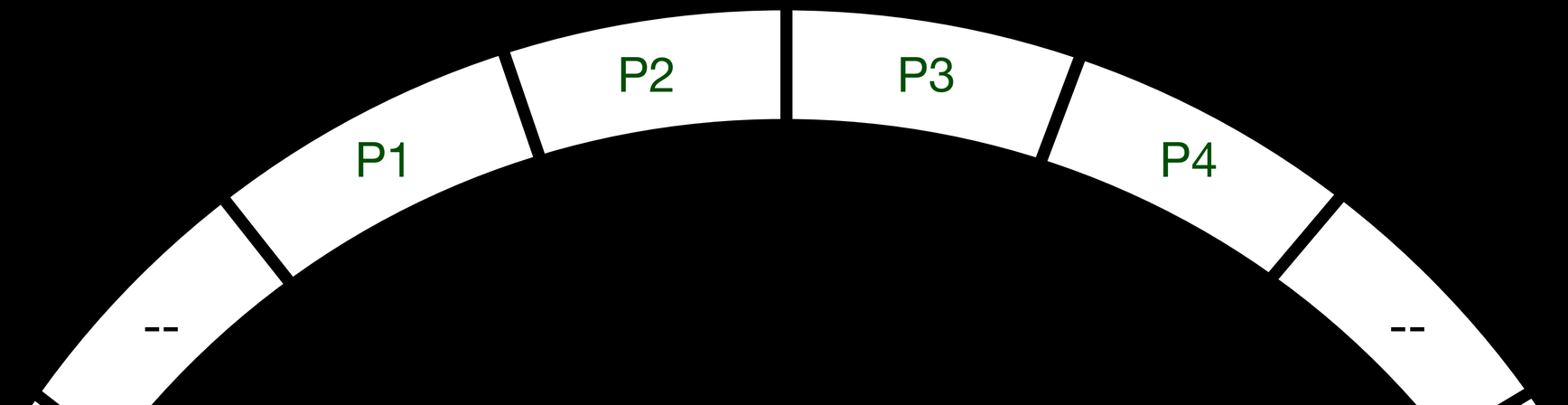

Application
QUIC
Socket
UDP
Interface Driver
recvmsg()
IP Packet to UDP socket

User-space UDP



QUIC

UDP/IP



Measurements

Speedtest download over QUIC

Home Wi-Fi

iPad, running with sockets and with user-space UDP

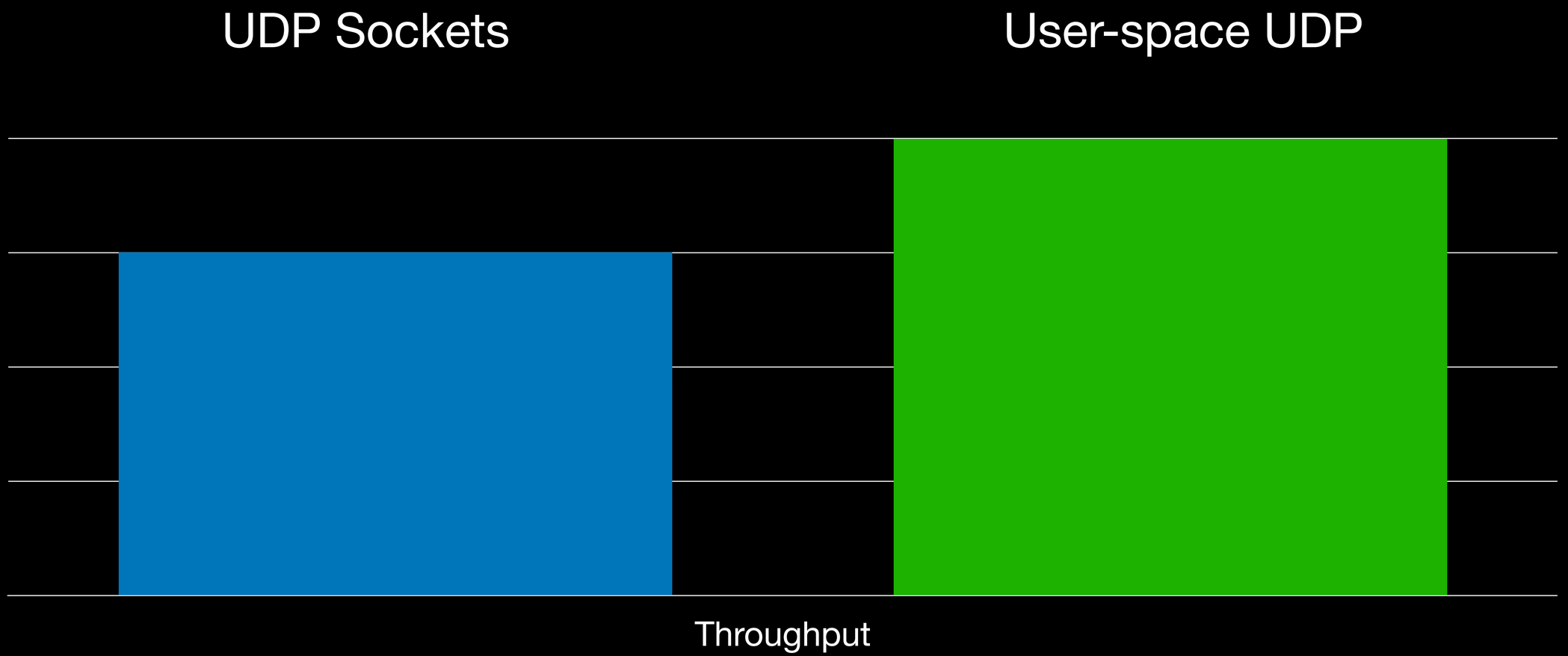
UDP Sockets

35K calls/sec

User-space UDP

15K calls/sec

57% better


UDP Sockets
User-space UDP
Throughput

Automatic failover

Happy Eyeballs

IPv6 / IPv6 address racing, as defined in RFC 8305

Not just for TCP!

- QUIC stacks can easily adopt Happy Eyeballs
- RTT heuristics should use protocol-specific caches

QUIC vs TLS racing

QUIC protocol racing with TLS

- Lower level than HTTP/3 -> HTTP/2 fallback
- Shared heuristics for detecting QUIC brokenness
- How will this generically work for non-HTTP applications?

Can we go further?

Using QUIC... for *everything*


MASQUE

iCloud Private Relay

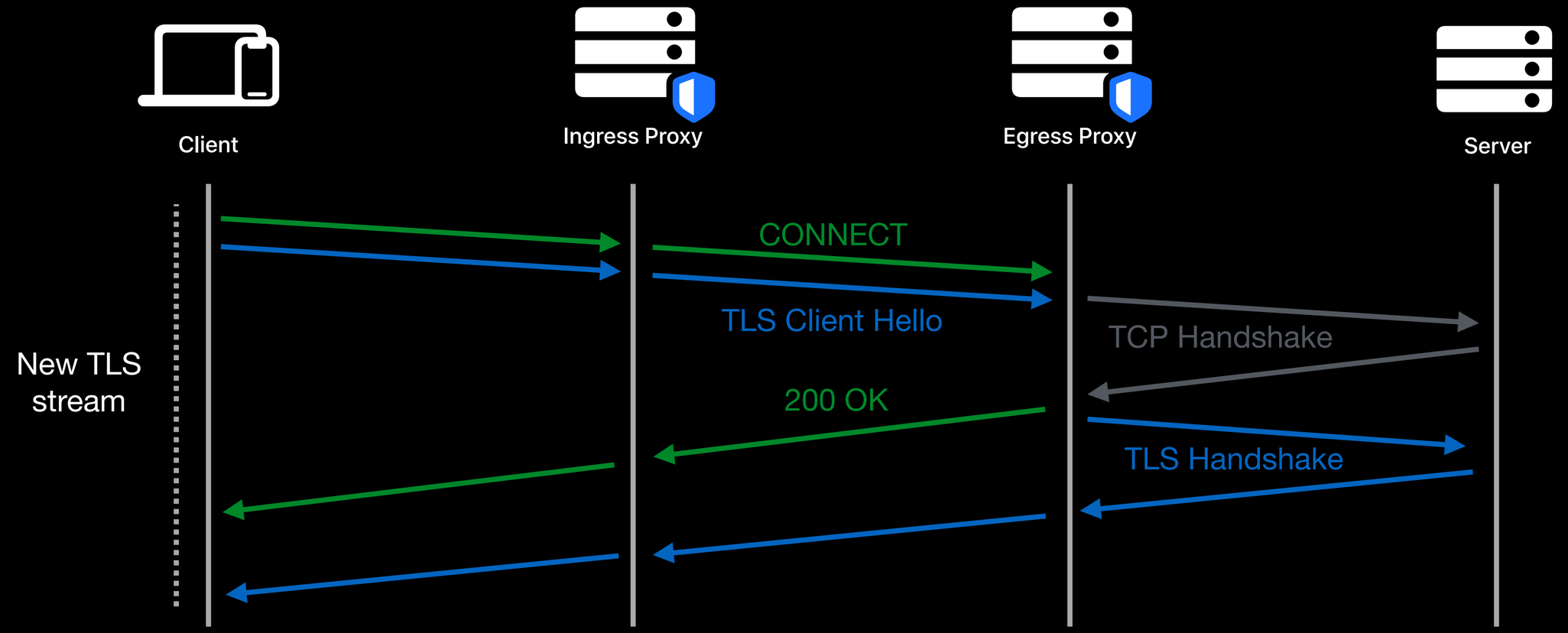
iCloud Private Relay uses MASQUE proxies

- HTTP/3 proxy built into the networking stack

Lots of traffic now uses QUIC

- All traffic in Safari (web browser)
- DNS
- Insecure app traffic

QUIC proxies



Comparison to other proxies

Secured with TLS 1.3

Multiplexing, connection reuse

Ability to proxy reliable and unreliable packets in one connection

- SOCKS and HTTP/1 don’t support multiplexing
- HTTP/2 proxying is not prevalent, doesn’t have fully independent streams

Comparison to traditional VPNs

DNS can be cached in the egress proxy

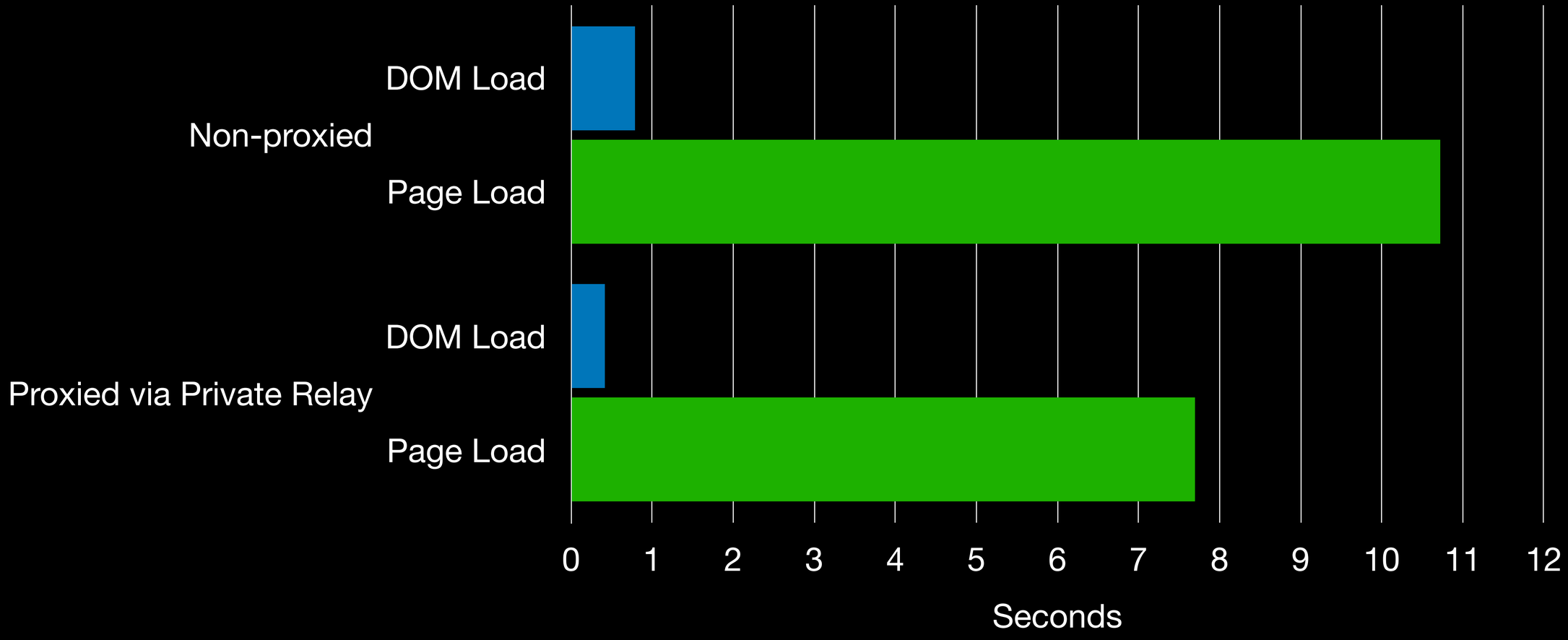
All connections get “fast open”

All connections get QUIC congestion control and loss recovery

Efficiently extends to multiple hops

Sample measurement

At home Wi-Fi test of cnn.com page load, using Safari Web Inspector



Stretching the boundaries

Proxy connections push limits that normal browser flows don’t hit

- Longer connection lifetime
- More streams being used (1000s)
- Longer stream lifetime
  - Each application connection maps to a new QUIC stream

Conclusion

QUIC is transforming networking on the client OS

We’re only starting to see what app adoption will look like

Proxying is a great way to get an early glimpse

How will systems and networks change once QUIC is more prevalent than TCP?