

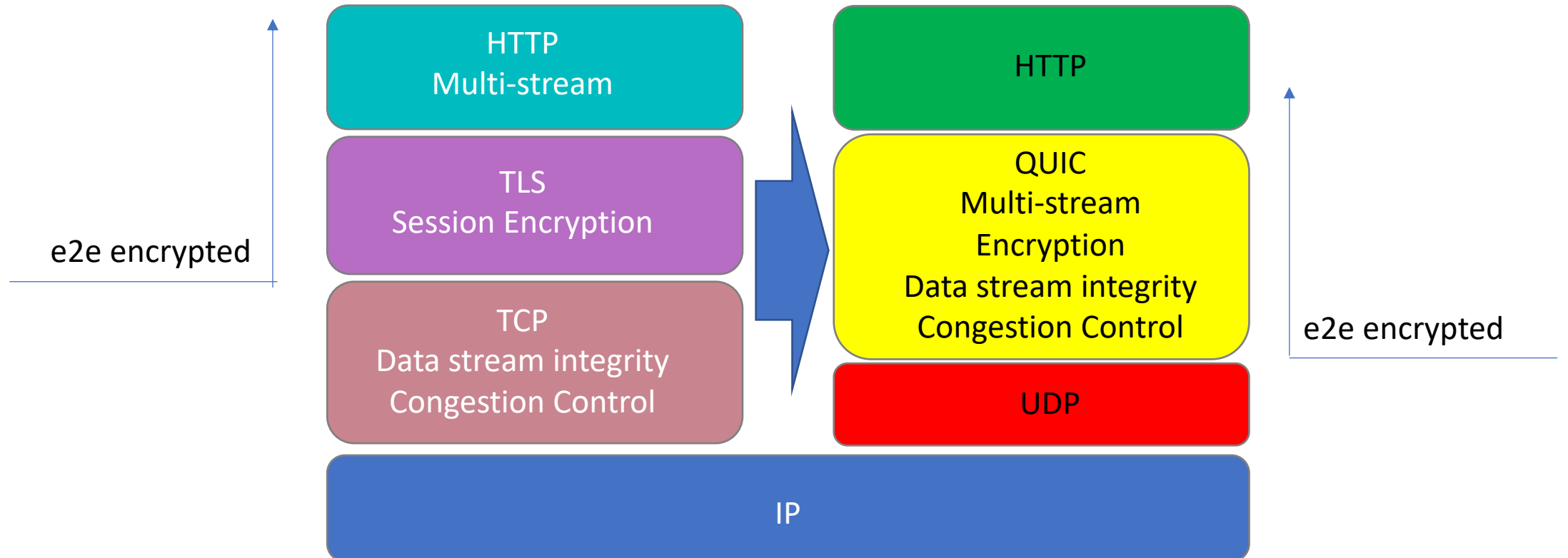
# A Quick look at QUIC

Geoff Huston

# QUIC is...

HTTP/2

QUIC  
HTTP/3



# TCP is..

A transport protocol that constructs a reliable full duplex adaptive streaming service constructed on top of an unreliable IP datagram service

- Uses a coordinated state between the two end systems without any network intervention or mediation
- Uses a sliding window to allow lost data to be resent
- Uses ACK-clocking to regulate the sending behaviour to match network path capacity estimate
- Has been tweaked to support (\*to some degree) multi-stream and RPC transport models

# TCP isn't...

- Fully independent of the underlying platform's transport services
- Fully multi-stream (it has head of line blocking)
- Free from on-the-wire network intervention (TCP control parameters are sent in the clear)
- Has e2e encryption as a second step / afterthought
- Everything for everyone – it relies on the application to perform data framing and in-band control

# QUIC is...

Constructed upon a transport level framing protocol that offers applications access to the basic IP datagram services offered by IP through the use of UDP

All other transport services (data integrity, session control, congestion control, encryption) are shifted towards the application. A platform may provide a QUIC API, but the application can also provide its own service

So much more than “encrypted TCP over UDP”:

Support for multi-stream multiplexing that avoids head-of-line blocking and exploits a shared congestion and encryption state

Faster - Combines transport and encryption setup exchange in a single 3-way exchange

Customisable - QUIC implementations can use individual flow controllers per flow

QUIC places its transport control fields inside the encryption envelope, so QUIC has minimal exposure to the network

Supports record and RPC service models as well as streaming and datagram

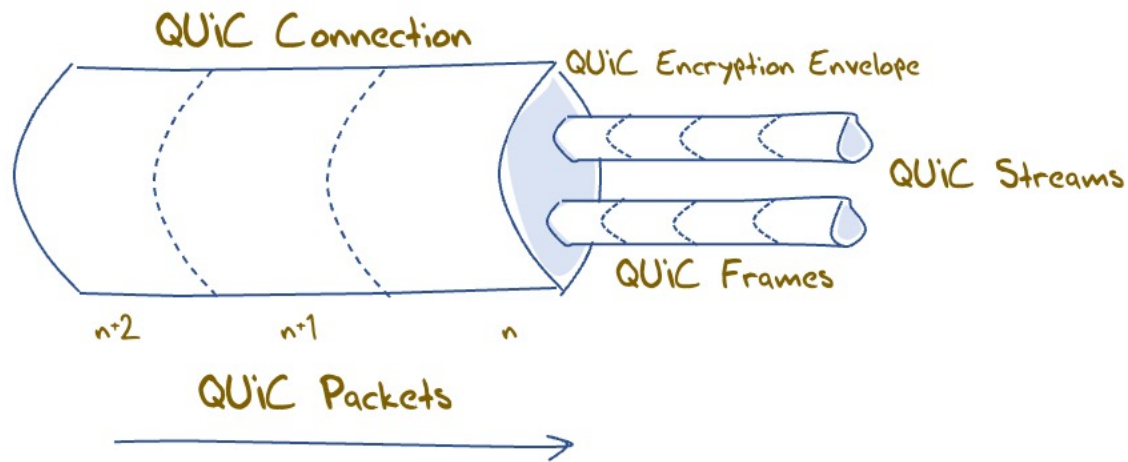
# QUIC is address agile

- NATs are hostile to QUIC because of the outer UDP wrapper
- A NAT may rebind (shift the externally visible address/port of a host during a session), as NATs are not generally aware of UDP streaming states
- QUIC uses a persistent “connection ID”
  - If a host receives a QUIC frame with the same connection ID and a new IP address / port it will send a challenge by way of a random value that should be echoed back. This is all performed within the e2e encryption envelope. That way QUIC can map into new address/port associations on the fly

# QUIC also...

- Is IP fragmentation intolerant – QUIC uses PMTUD, or defaults to 1,200 octet UDP payloads
- Never retransmits a QUIC packet – retransmitted data is sent in the next QUIC packet number – this avoids ambiguity about packet retransmission
- Extends TCP SACK to 256 packet number ranges (up from 3)
- Separately encrypts each QUIC packet
- May load multiple QUIC packets in a single UDP frame

# QUIC flow structuring



A QUIC connection is broken into “streams” which are reliable data flows – each stream performs stream-based loss recovery, congestion control, and relative stream scheduling for bandwidth allocation

QUIC also supports unreliable encrypted datagram delivery



# QUIC and RPC

- By associating each RPC request/reply with a new stream QUIC can support asynchronous RPC transactions using reliable messaging
  - This can handle lost, mis-ordered and duplicated RPC messages without common blocking or throttling

# QUIC and Load Balancing

- This assumes that a front-end load balancer is capable of performing load balancing on UDP flows using the UDP connection 5-tuple
- If the remote end performs NAT rebinding the load balancer will be thrown by this shift, and it has no direct visibility into the e2e session to uncover the connection ID
- Using UDP to carry sustained high-volume streams may not match the internal optimisations used in server content delivery networks
- If we really want large scale QUIC with front end load balancing and we still need to tolerate NATs then we will need to think about how the end point can share the connection ID state with its front end load balancer, or terminate the QUIC session in the front end and use a second session to the selected server


# QUIC and DOS

- Very little lies outside the encryption envelope in QUIC
- Which means all incoming packets addressed to the QUIC port need to be decrypted
- But the session uses symmetric crypto so the packet decode overhead is far smaller than the asymmetric load
- Its not the best answer, but its not disastrous either!

# Looking for QUIC

- At APNIC we use Ads to perform large scale measurements of network service capabilities as seen by users
  - IPv6 deployment
  - DNSSEC validation
  - Fragmentation
- Can we use this measurement platform to see the level of use of QUIC in today's network?

# Setting up QUIC

- Server:
  - nginx v1.21.7 with QUIC functions included
- DNS:
  - Set up an HTTPS record for each URL with value: **alpn="h3"**
- Content:
  - **Alt-Svc: h3=":443"** 

 (This second method requires a subsequent query to allow the client to use the Alt-Svc capability. We perform a delayed second query for this URL in the measurement experiment)

# Two QUIC triggers, two behaviours

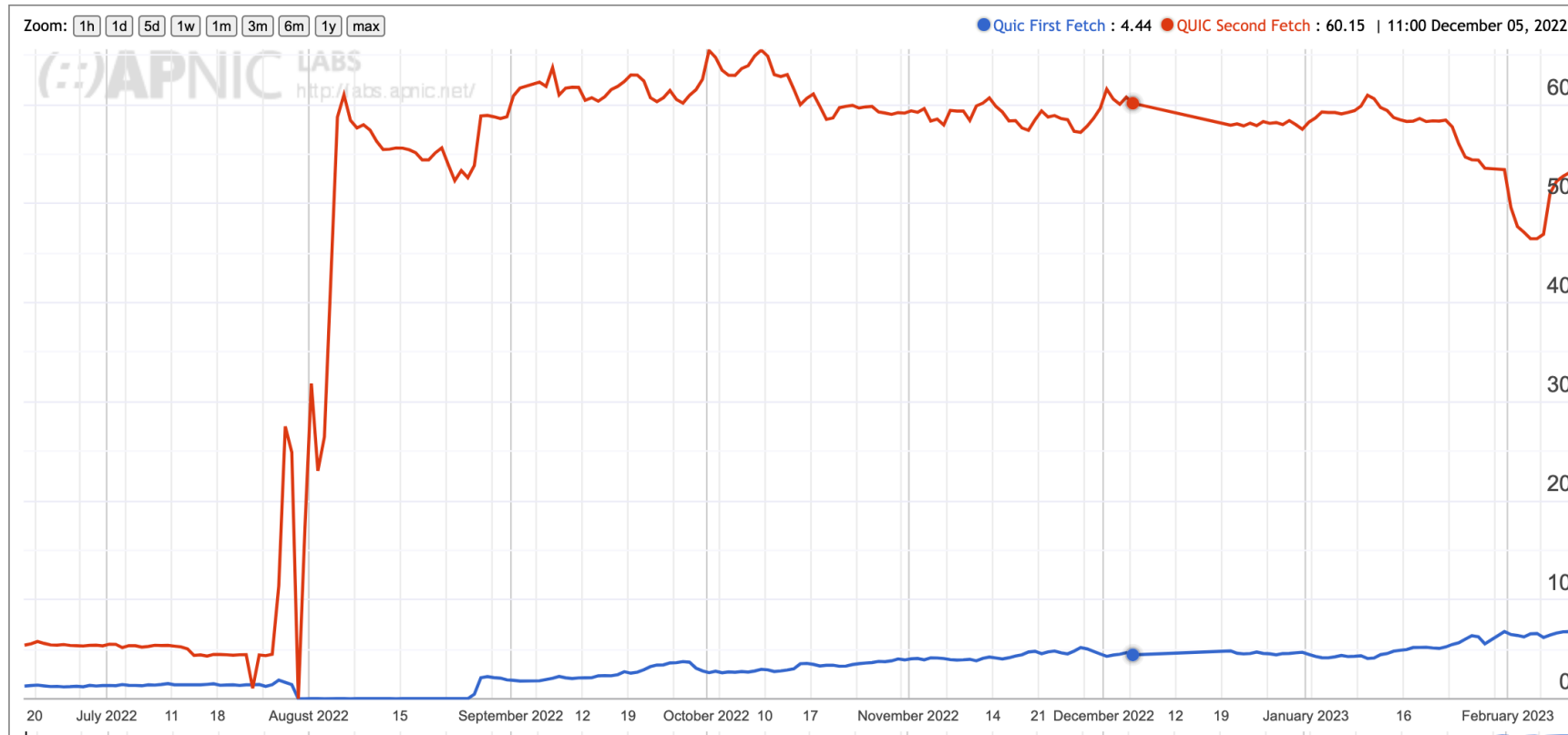
- The Safari browser (V16) is triggered by the presence of the DNS HTTPS record with an “alpn” value of “h3”
  - Once established, the browser client expects the server to keep the QUIC session open. If the session is closed “prematurely” Safari will drop have to HtTP/2 over TCP/TLS
- What’s “premature” in seconds?
  - I dunno!
  - More than 1 second. Less than 65 seconds.
  - We use a server session keepalive interval of 20 seconds and that seems to work

# Two QUIC triggers, two behaviours

- The Chrome browser is triggered by the content directive within the delivered page
  - But this creates an issue with HTTP/2 and persistent connection support
  - When the browser performs the followup fetch after the initial fetch, the connection is likely still open. The browser will prefer to use the open connection rather than opening up a new QUIC connection, despite the inclusion of a content directive. So this in-content signalling is not all that effective as a QUIC trigger.
- How long should we wait before performing the second query? Or how long should we set the server's keepalive persistence timer?
  - I dunno!
  - More than 1 second. Less than 65 seconds.
  - We use a keepalive interval of 20 seconds and that seems to work

# Measuring QUIC use

## Use of HTTP/3 for World (XA)

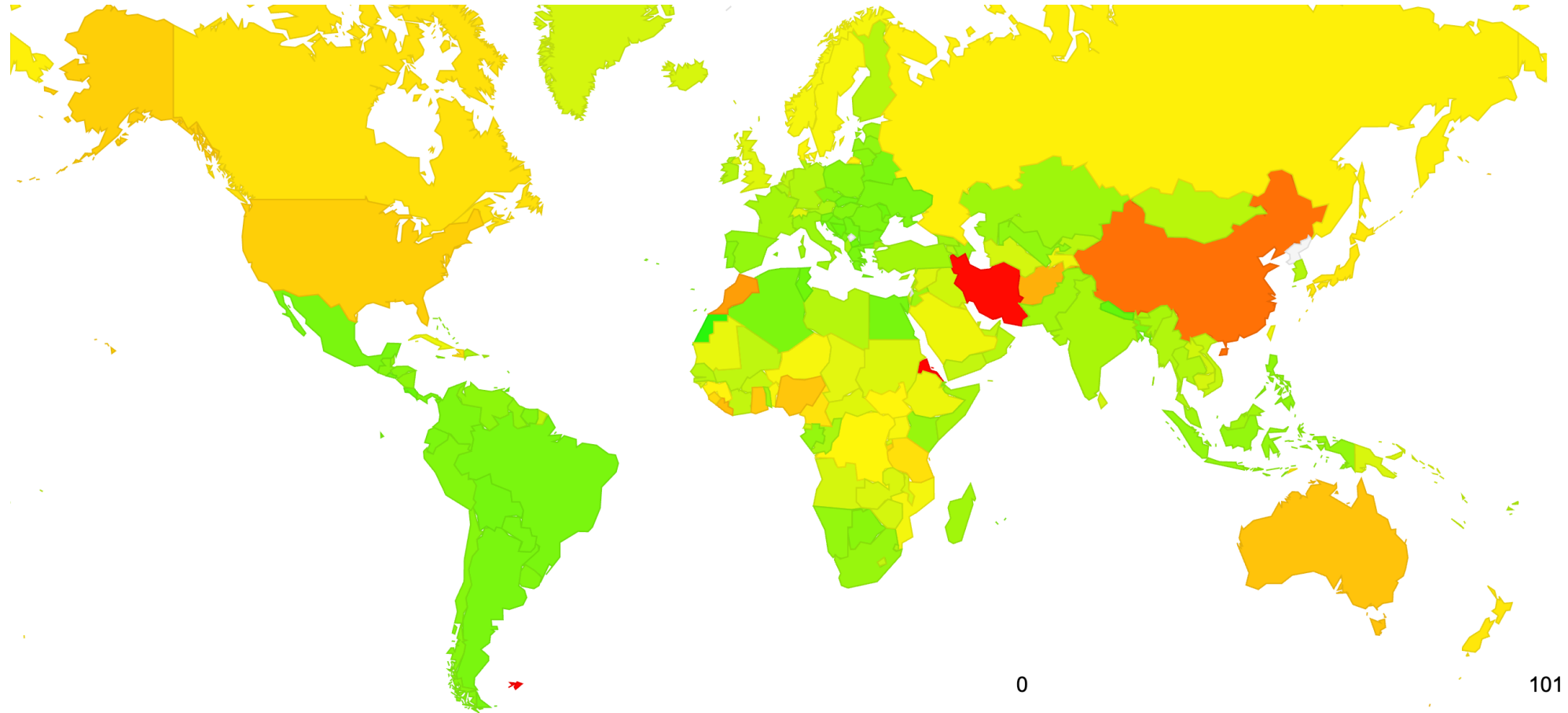


2<sup>nd</sup> and subsequent fetches

First fetch



# Measuring QUIC Use



# Why?

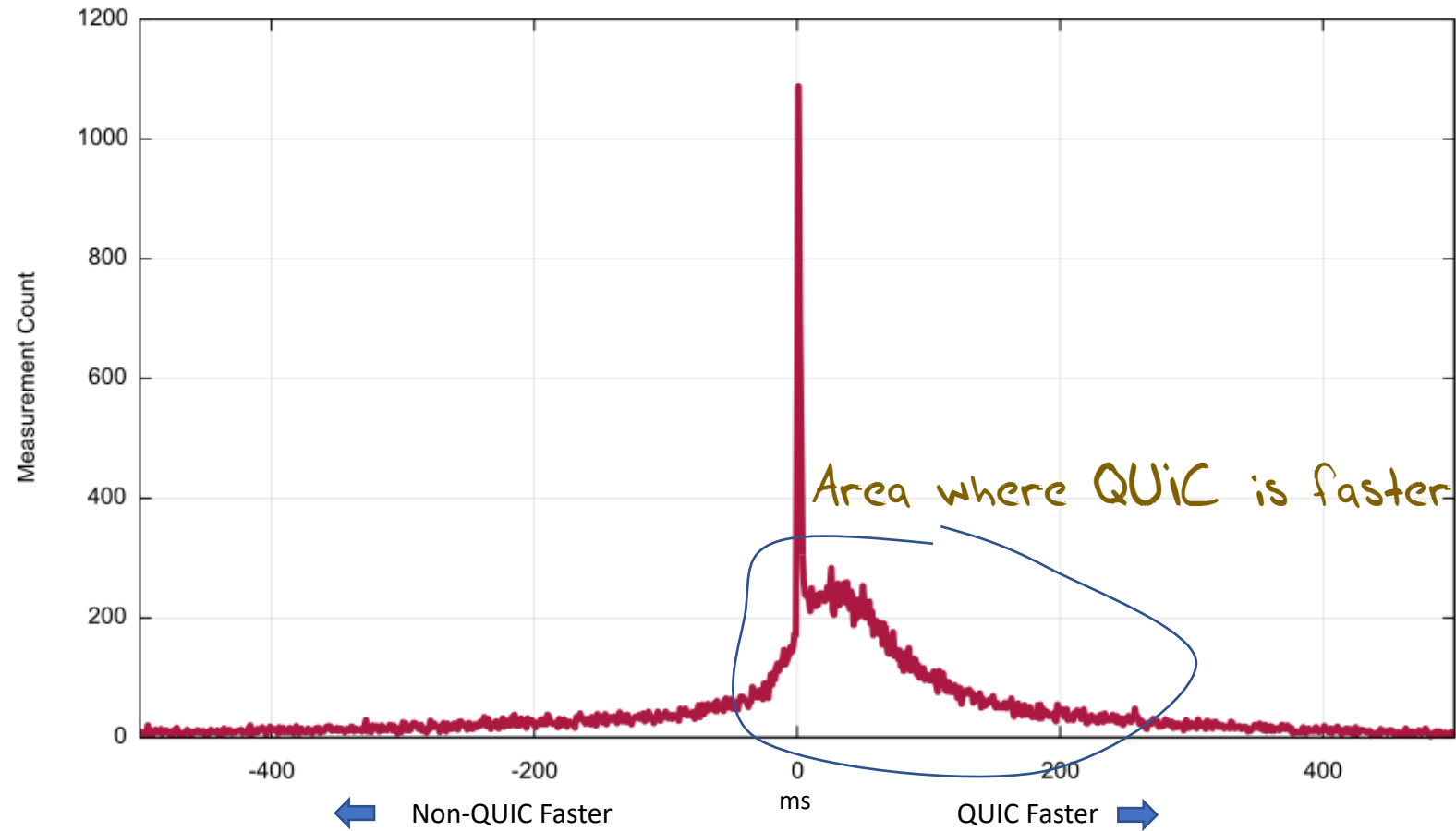
- If both Safari and Chrome support QUIC these days then why is the measurement number not closing in on 100%?
  - I suspect its about many enterprise environments blocking UDP port 443
  - And some ISPs
  - And of course the Great Firewall of China

# Is QUIC Faster?

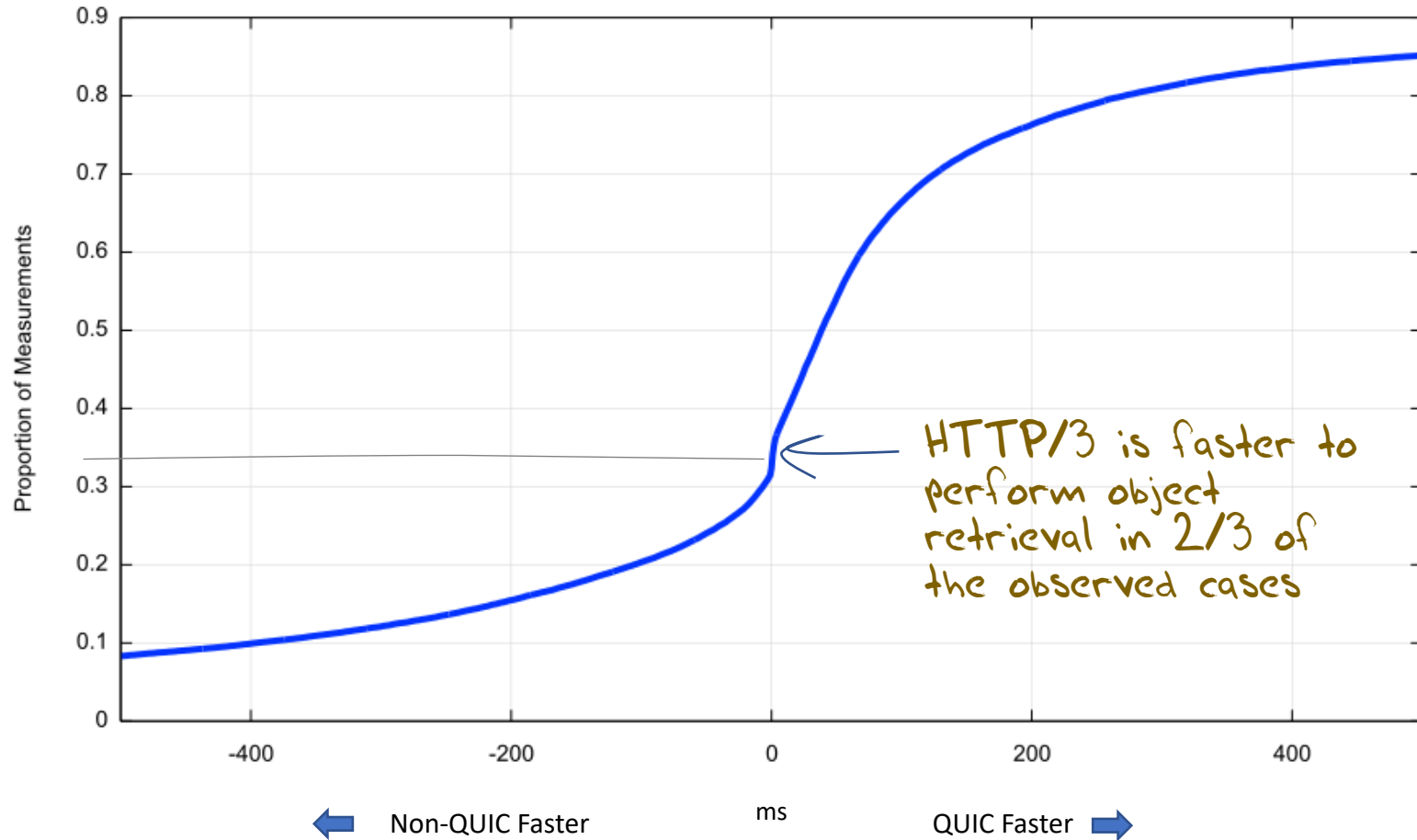
Let's compare the user-measured time to load an object using HTTP/2 and the same user's measured time to load the same object using HTTP/3

- There are a number of variables in the user time measurement, including varying time penalties relating to the internal task scheduling within the browser, but these individual factors should be cancelled out over a large enough sample set

# Is QUIC faster?



# Is QUIC faster?



Thanks!

**Ongoing HTTP/3 Measurement Report at APNIC Labs:  
<https://stats.labs.apnic.net/quic>**