The challenges of 0-RTT in IETF QUIC

ianswett@, fayang@

Terminology

gQUIC - The experimental protocol developed by Google.

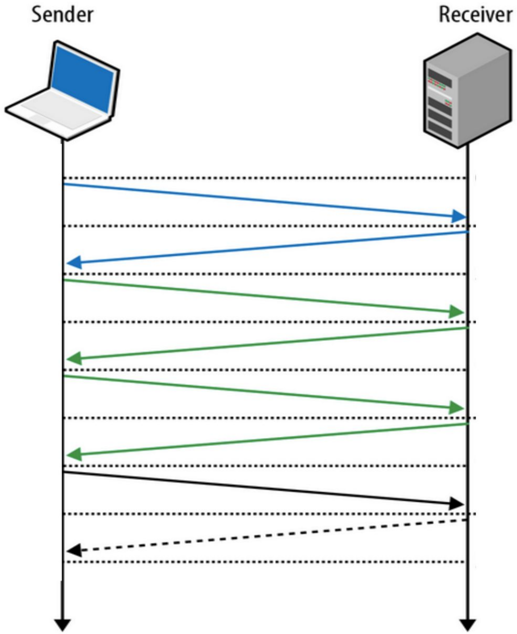
Uses ‘QUIC Crypto’ by Adam Langley.

QUIC or IETF QUIC - The protocol standardized by the IETF

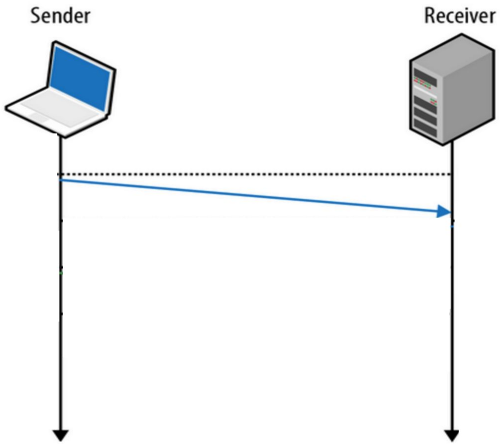
HTTP/3 - HTTP over IETF QUIC

Why 0-RTT? Immediately send an HTTP Request

TCP + TLS
2-3 RTTs



QUIC
0-1 RTTs



A 0-RTT Handshake

QUIC Encryption Levels

Initial - Basically Obfuscation as the keys are in the RFC

Handshake - Keys derived from Client and Server Initials

0-RTT - Keys exported on the client for sending application data in the first flight

1-RTT - Forward Secure keys used after the handshake completes

- **0.5-RTT** - Term for 1-RTT data sent by the server before handshake completion

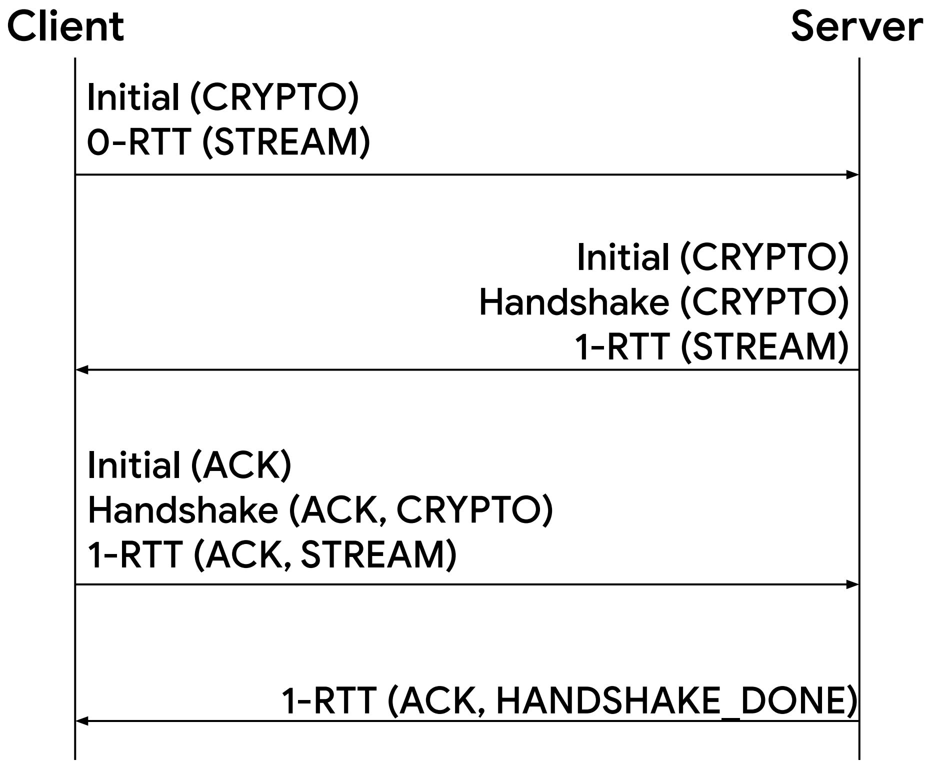
QUIC Packet Number Spaces

Only **Initial** packets can ACK **Initial** packets

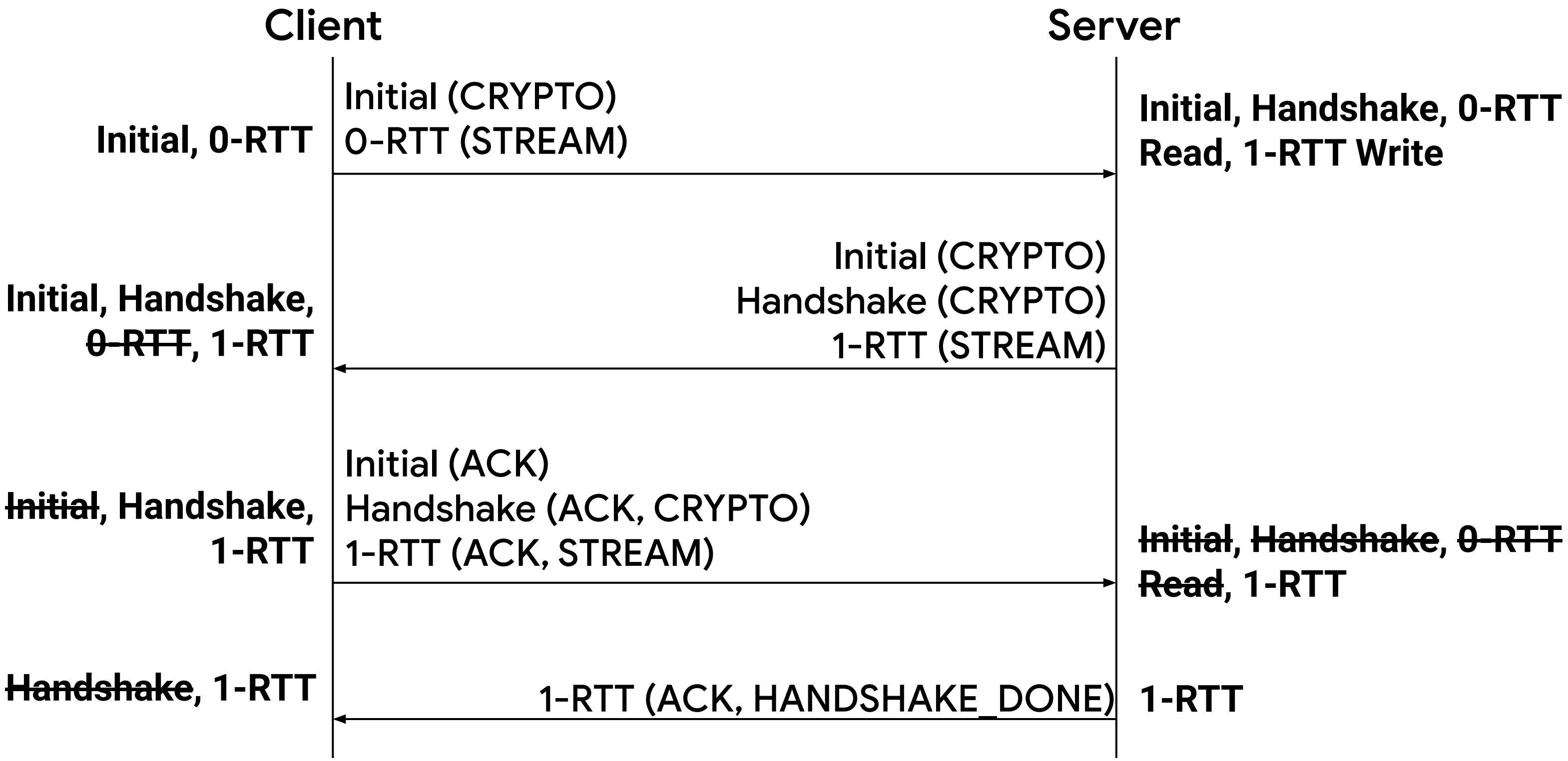
Only **Handshake** packets can ACK **Handshake** packets

Only **1-RTT** packets can ACK **0-RTT** or **1-RTT** packets

What Successful 0-RTT looks like



Available Encryption and Decryption Keys



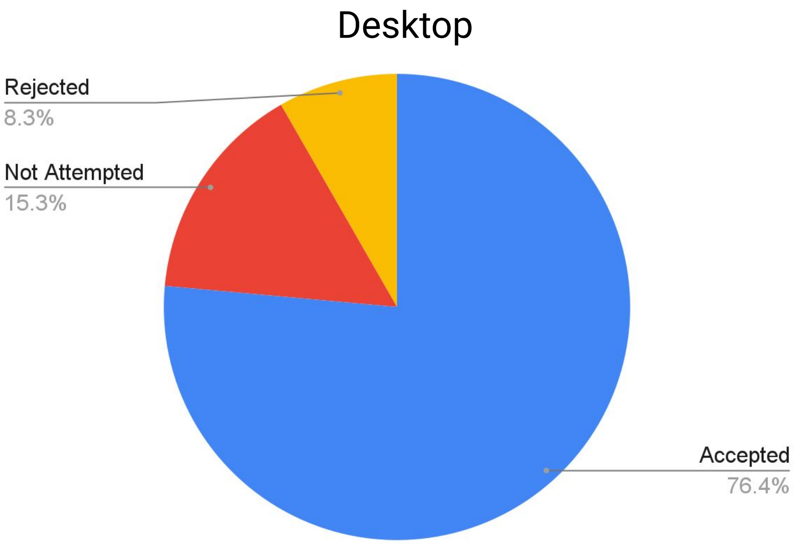
0-RTT Restrictions

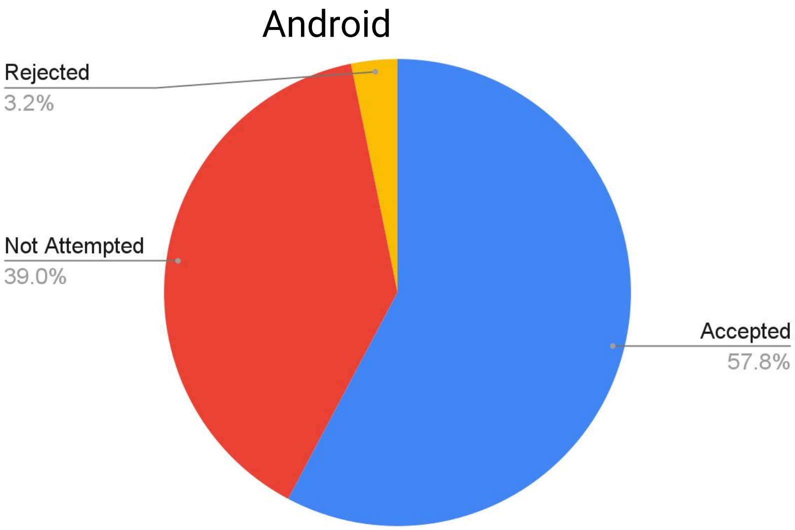
- Have connected to the server 'recently'
  - And persisted across 3 layers:
    - HTTP/3 SETTINGs, QUIC Token and TLS NewSessionTicket
- Can only send safe HTTP methods
  - GET, HEAD, OPTIONS, TRACE
- HTTP/3 SETTINGs can't change

Some bonus challenges

- QUIC's 3x amplification limit before address validation
    - To send a large response, client IP must not change
- Chrome doesn't persist NewSessionTickets to disk
  - gQUIC Server Configs could be persisted
- Up to 3 Packet Number spaces at once
  - Limited knowledge of which keys the peer has
- Chrome blocks using 0-RTT keys on certificate revalidation
  - Resumption can be almost as fast if the RTT is small.
- 0-RTT packets can be reordered
  - Servers need to decide whether to buffer them

0-RTT Success Rates





Initial Experiment (11/2020)

0-RTT had neutral mean latency for Search and slower tail (90%+) latency.

A Note on Data

Experiments are randomized in Chrome

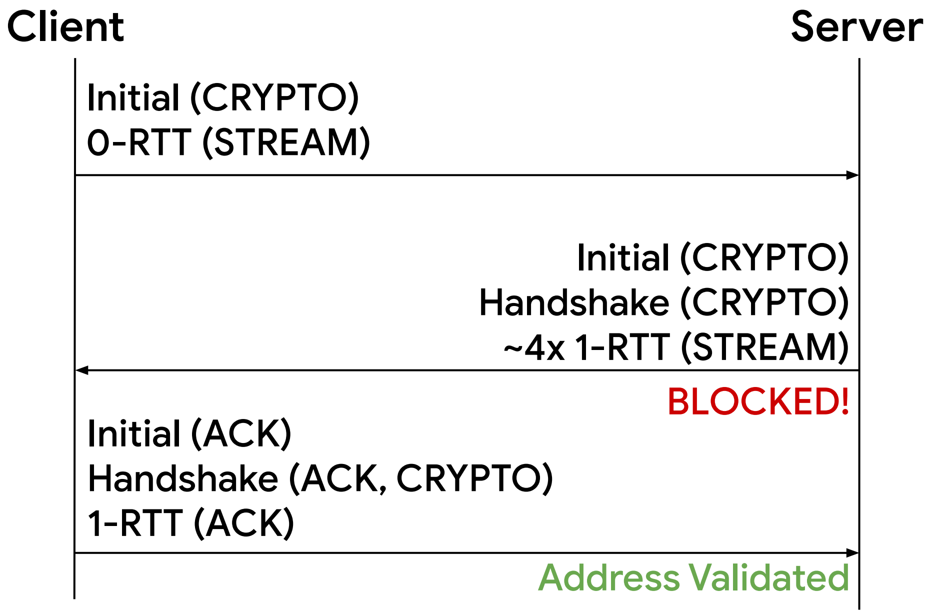
Even if QUIC or 0-RTT don’t work, the data from those users are included

First: Amplification Limit

Amplification Limit

0-RTT experiment **10x** more likely to be throttled by the 3x anti-amplification limit

Once limited, the server waits for the client to unblock it.



Fix: Address Validation Token

IETF QUIC allows including a ‘Token’ in the Client Initial

- Server decrypts the Token
- Validates that the client’s address is unchanged

Result: Didn’t move the metrics much… A bit closer to neutral

Second: PTOing at Correct Encryption Level(s)

Increased Handshake Timeouts

~2x increase in pre-handshake client NETWORK_IDLE_TIMEOUT

=> Client hasn’t processed anything from the server in 4 seconds (Chrome)

Our Issue

The 0-RTT response is large, server becomes blocked by the amplification limit

So PTO is not armed

Our optimization bundled other Initial data with an Initial ACK

Rearmed the PTO for the future, then sent more 0.5 RTT data

Never send Handshake data, **deadlock**

Key PTO Fixes

If the PTO would have fired, execute it before other sending

When the PTO fires, send in multiple packet number spaces

Always PTO Initial and Handshake data before 0-RTT or 1-RTT

Result: Fixed the pre-handshake NETWORK_IDLE_TIMEOUT increase!

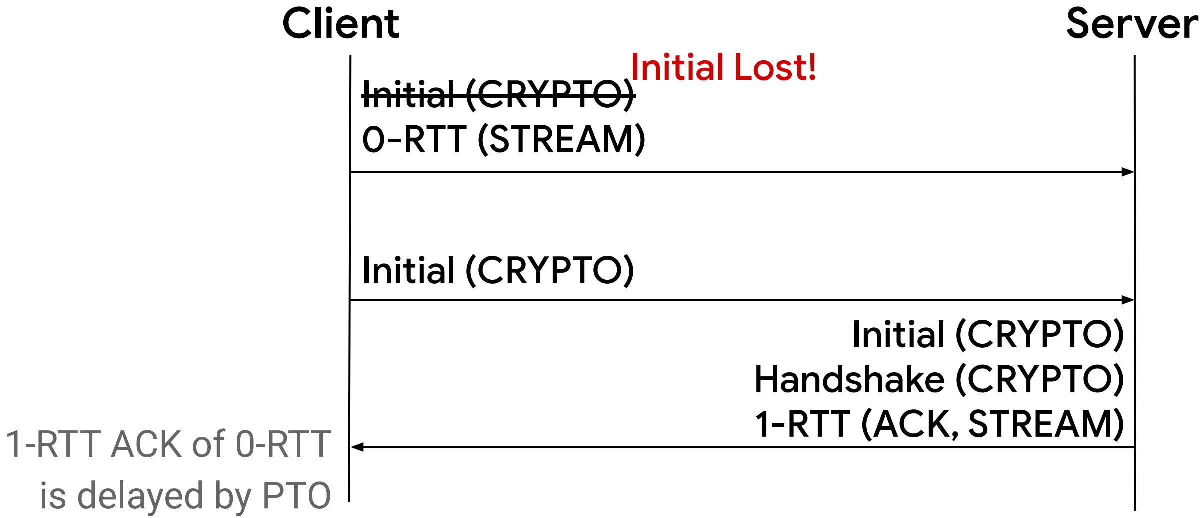
…. Still not faster

Third: Inflated RTT Samples

Why could RTT be inflated

Keys might not be available to process and ACK the packet

- Queue undecryptable packet
- Later, keys become available, packet processed, ACK sent



Why do Inflated RTTs slow the handshake?

Inflated RTT samples increase the [Probe Timeout]

```
PTO_delay = SmoothedRTT + 4 * RTTVariation + max_ack_delay
```

Connections start with no RTT samples, default PTO of 1 second

If a packet is lost, no progress until the probe timeout fires

ie: 1s RTT sample = 3s PTO timeout!

Fixes

Send delayed ACK based on packet receipt time, not packet processing time

Optimization

Send packets of higher packet number spaces when a packet can cause peer to generate new keys

Ie: Server retransmits INITIAL packet, send HANDSHAKE and 0.5 RTT packets.

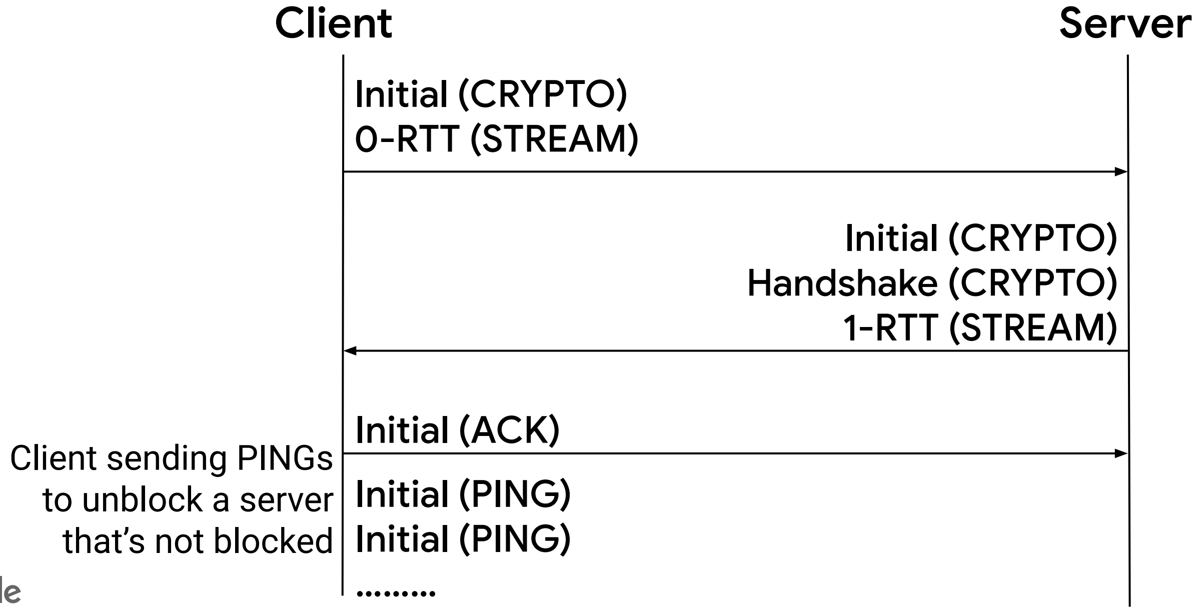
Result: Further reduction in handshake timeouts

Fourth: Async Client Bug

Client sends INITIAL pings until timeout

Traces showed clients never sending HANDSHAKE packets

- But the ServerHello had been acknowledged



Client never derives Handshake keys

Luckily, we got an external report with a Chrome net-log attached

- The client **never** derives handshake keys.

Chrome starts certificate verification in parallel

- When verification finishes after receiving ServerHello, never get Handshake keys

Finally: Little issues and Optimizations

Little Issues

Marked a packet (and its data) as retransmitted when it wasn't

Error in size calculation => Coalesced packet that was too big, failed sending

Processing buffered packets in order, stopped if one fails to decrypt

Delaying PTO when sending 0-RTT packets

Little Optimizations

Delay the server's first ACK until it can be bundled with the ServerHello

Coalescing pending ACKs of other packet number spaces

Coalescing HANDSHAKE and NewSessionTicket with ServerHello

Recap and Results

Google

Lessons Learned

Tooling is critical

- Packet traces enabled root-causing many bugs

Sharing code with gQUIC was sometimes helpful

- But sometimes introduced subtle bugs due to differences

Getting PTO right during the handshake is difficult

Finally...

Chrome Desktop (-0.3%, -0.6%@99%)

Chrome Android (-0.3%, -0.6%@99%):

0-RTT default enabled Sept 2021 in Chrome M95!

Further fixes have landed since

But wait, I thought 0-RTT would save an RTT?

Chrome pre-connects, eliminating handshake latency

Not every search requires a new connection

There are more bugs and optimizations to be found

Thanks!

Chromium QUIC Code
cs.chromium.org

Google