The QUIC Transport Protocol

- Design and Internet-Scale Deployment

Chen, Zixuan

OUTLINE

0

Outline

1 **INTRODUCTION**

一段描述的语言，不要太长，但是超过一行会比较美观。

2 **WHY QUIC**

一段描述的语言，不要太长，但是超过一行会比较美观。

3 **DESIGN & IMPLEMENTATION**

一段描述的语言，不要太长，但是超过一行会比较美观。

4 **EXPERIMENTS & EXPERIENCES**

一段描述的语言，不要太长，但是超过一行会比较美观。

INTRODUCTION

1

WHAT’s QUIC

QUIC (Quick Udp Internet Connection)

A new transport designed from the ground up to improve performance for HTTPS traffic and to enable **rapid deployment** and **continued evolution of transport mechanisms**.

QUIC replaces most of the traditional HTTPS stack: HTTP/2, TLS, and TCP (Figure 1).

QUIC is developed as a user-space transport with UDP as a substrate.

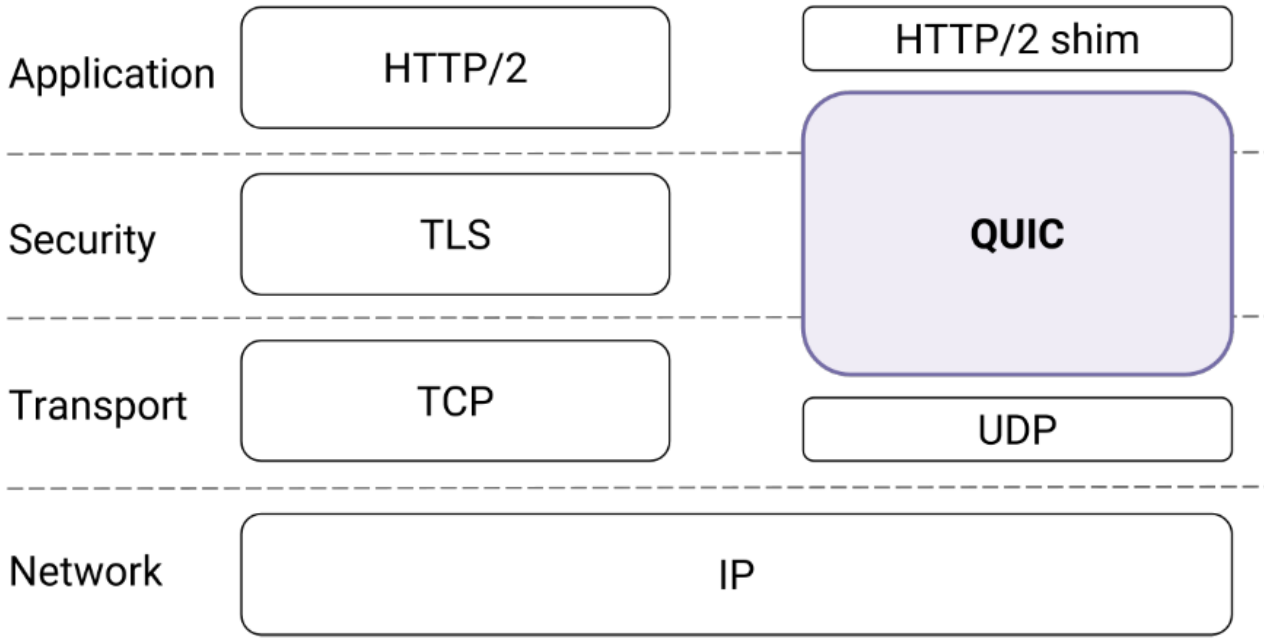


Figure 1: QUIC in the traditional HTTPS stack.

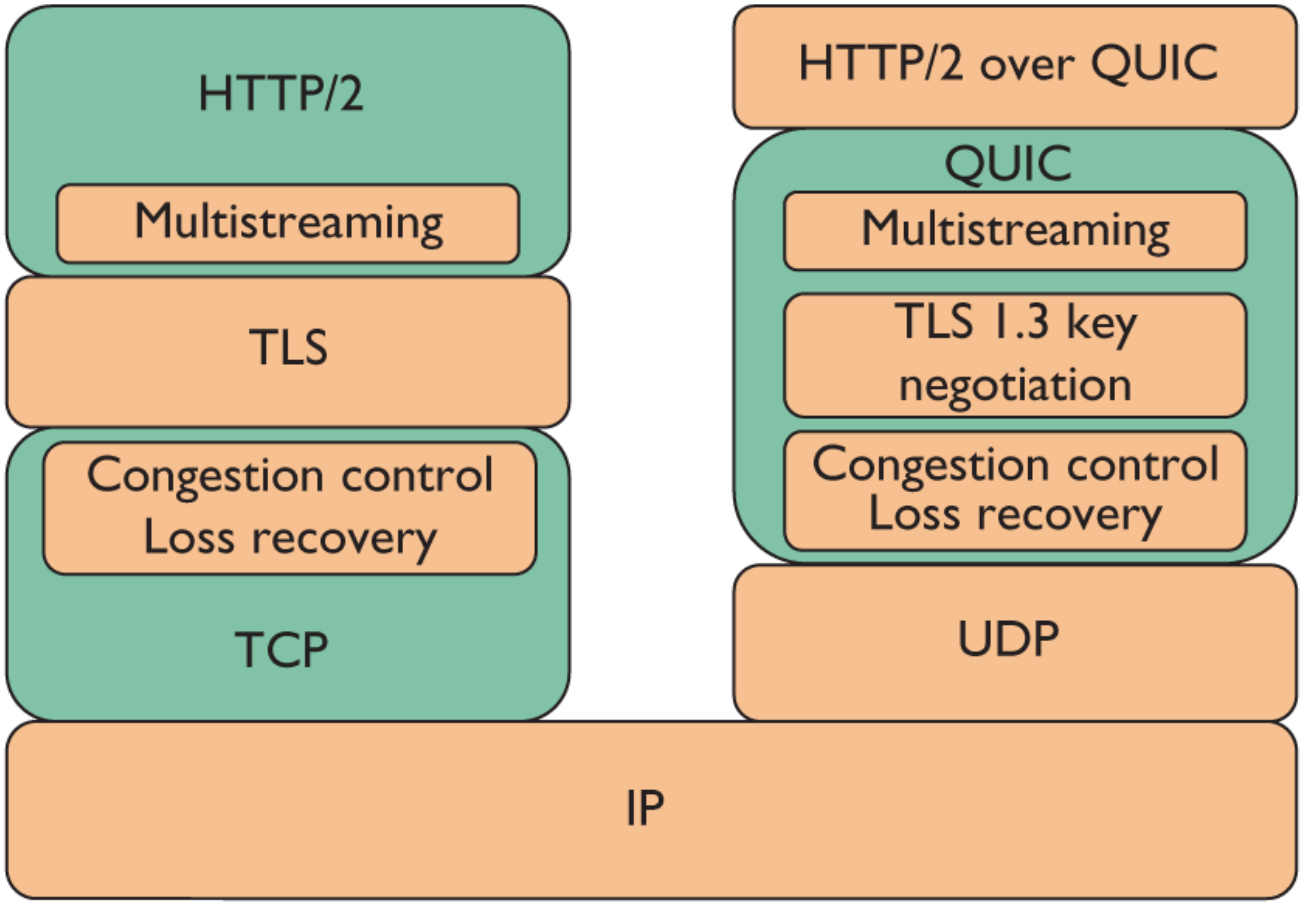
WHAT's QUIC

QUIC (Quick Udp Internet Connection)

A new transport designed from the ground up to improve performance for HTTPS traffic and to enable **rapid deployment** and **continued evolution of transport mechanisms**.

QUIC replaces most of the traditional HTTPS stack: HTTP/2, TLS, and TCP (Figure 1).

QUIC is developed as a user-space transport with UDP as a substrate.



本站开始支持 QUIC
https://halfrost.com/quic_start/

The Road to Deployment

QUIC has been globally deployed at Google on thousands of servers and is used to serve traffic to a range of clients including a widely-used web browser (Chrome) and a popular mobile video streaming app (YouTube).

On the server-side, our experience comes from deploying QUIC at Google's front-end servers, which collectively handle billions of requests a day from web browsers and mobile apps across a wide range of services.

On the client side, we have deployed QUIC in Chrome, in our mobile video streaming YouTube app, and in the Google Search app on Android. We find that on average, QUIC reduces latency of **Google Search responses by 8.0% for desktop users and by 3.6% for mobile users,** and reduces **rebuffer rates of YouTube playbacks by 18.0% for desktop users and 15.3% for mobile users.**

The Road to Deployment

As shown in Figure 2, QUIC is widely deployed: it currently accounts for over 30% of Google’s total egress traffic in bytes and consequently an estimated 7% of global Internet traffic [61].

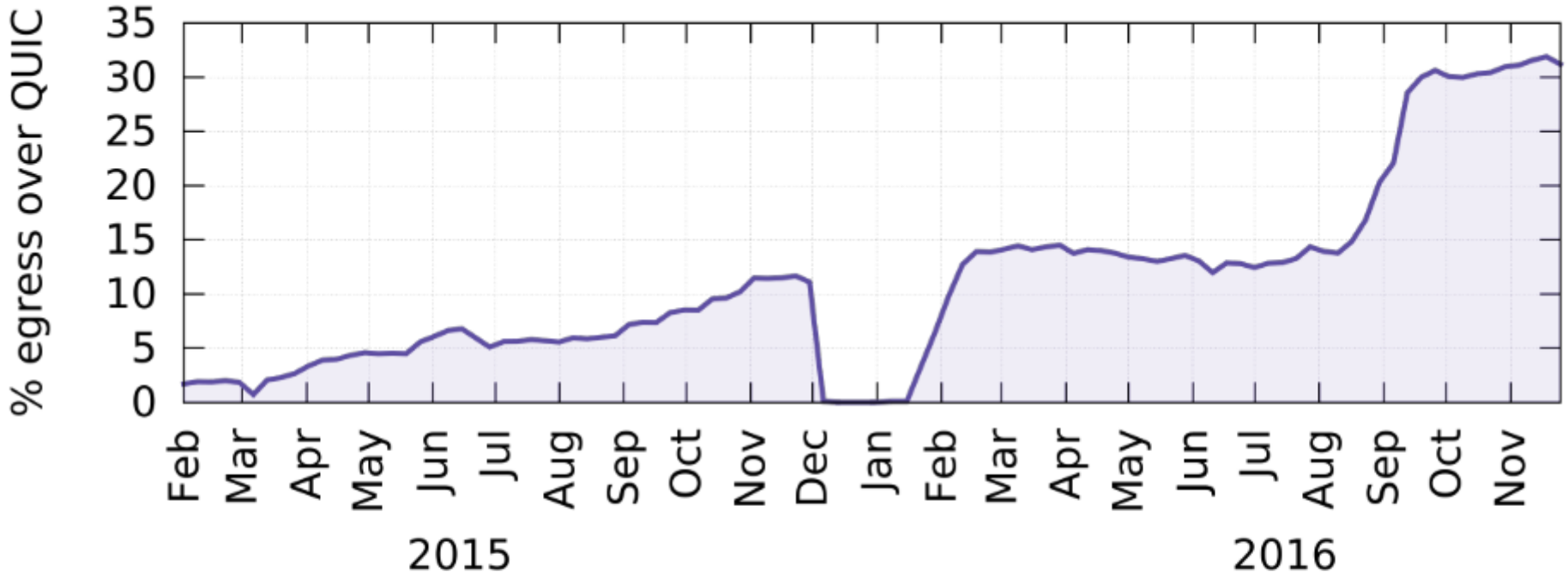


Figure 2: Timeline showing the percentage of Google traffic served over QUIC. Significant increases and decreases are described in Section 5.1.

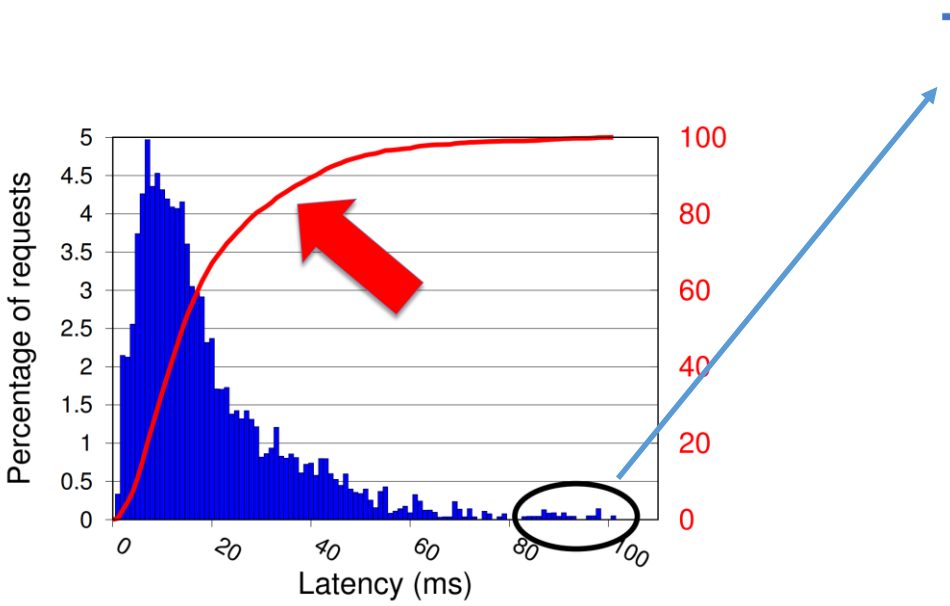
WHY QUIC

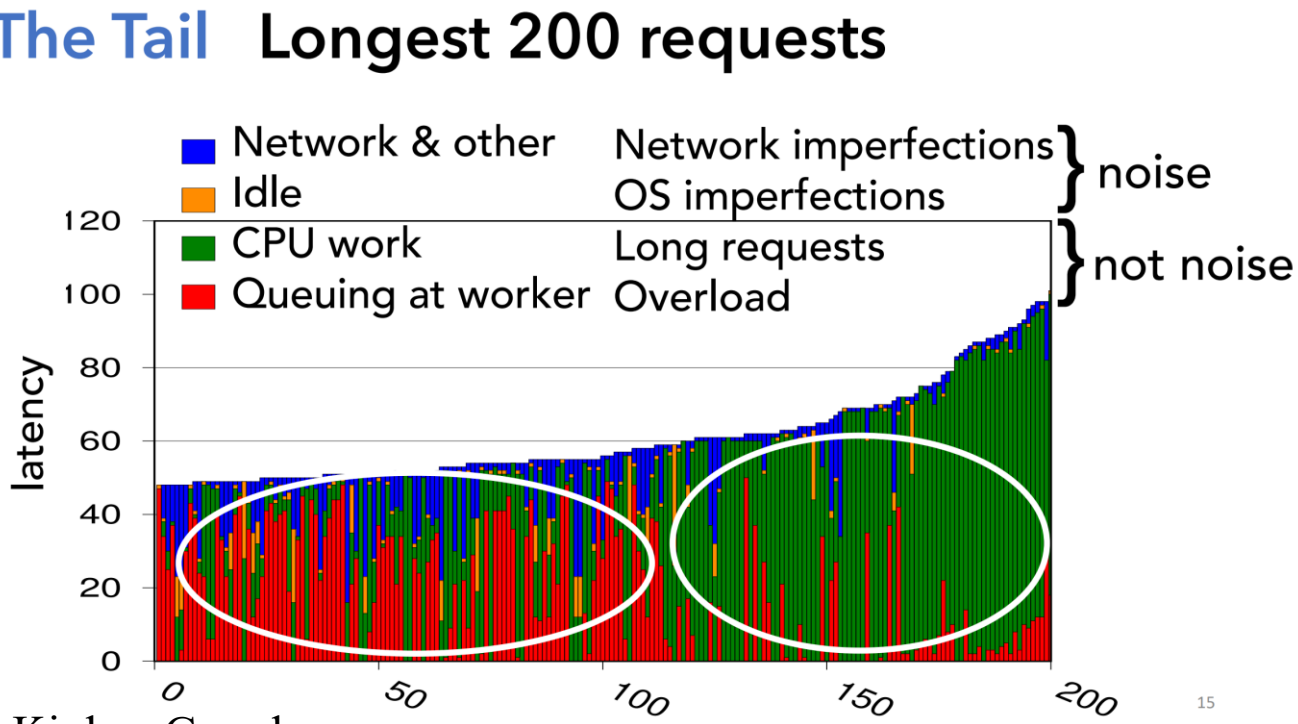
2

MOTIVATION: WHY QUIC

Growth in latency-sensitive web services and use of the web as a platform for applications is placing unprecedented demands on reducing web latency.

Web latency remains an impediment to improving user experience, and tail latency remains a hurdle to scaling the web platform.





Tail Latency

Measuring and Optimizing Tail Latency, Kathryn S McKinley, Google

https://cra.org/cra-wp/wp-content/uploads/sites/8/2018/04/VUTH-14-combined-slides.compressed.pdf

MOTIVATION: WHY QUIC

Shortcomings of TCP

- Protocol Entrenchment
- Implementation Entrenchment
- Handshake Delay
- Head-of-line Blocking Delay

Protocol Entrenchment

Middleboxes have accidentally become key control points in the Internet's architecture.
Firewalls tend to block anything unfamiliar for security reasons and Network Address Translators (NATs) rewrite the transport header, making both **incapable of allowing traffic from new transports without adding explicit support for them.**

Any packet content not protected by **end-to-end security**.

Modifying TCP remains challenging due to its **ossification by middleboxes**.
Deploying changes to TCP has reached a point of diminishing returns, where simple protocol changes are now expected to **take upwards of a decade** to see significant deployment.

Implementation Entrenchment

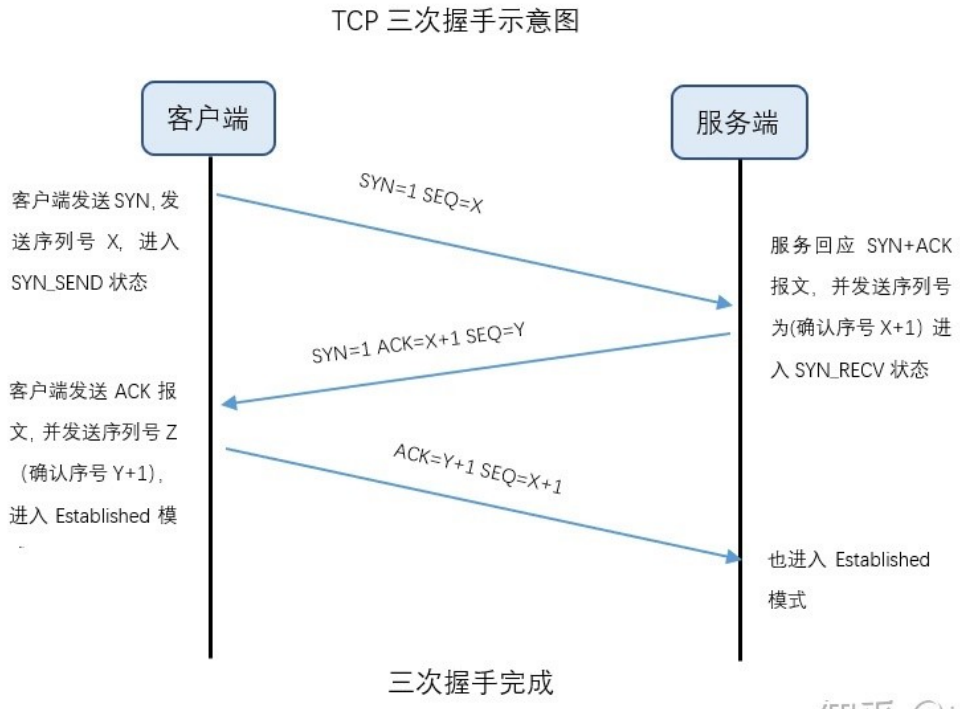
TCP is commonly implemented in the Operating System (OS) kernel. As a result, even if TCP modifications were deployable, pushing changes to TCP stacks typically **requires OS upgrades**. This coupling of the transport implementation to the OS limits deployment velocity of TCP changes; OS upgrades have system-wide impact and the upgrade pipelines and mechanisms are appropriately cautious [28].

OS upgrades at servers tend to be faster by an order of magnitude but can **still take many months** because of appropriately rigorous stability and performance testing of the entire OS.

This limits the deployment and iteration velocity of **even simple networking changes.**

Handshake Delay

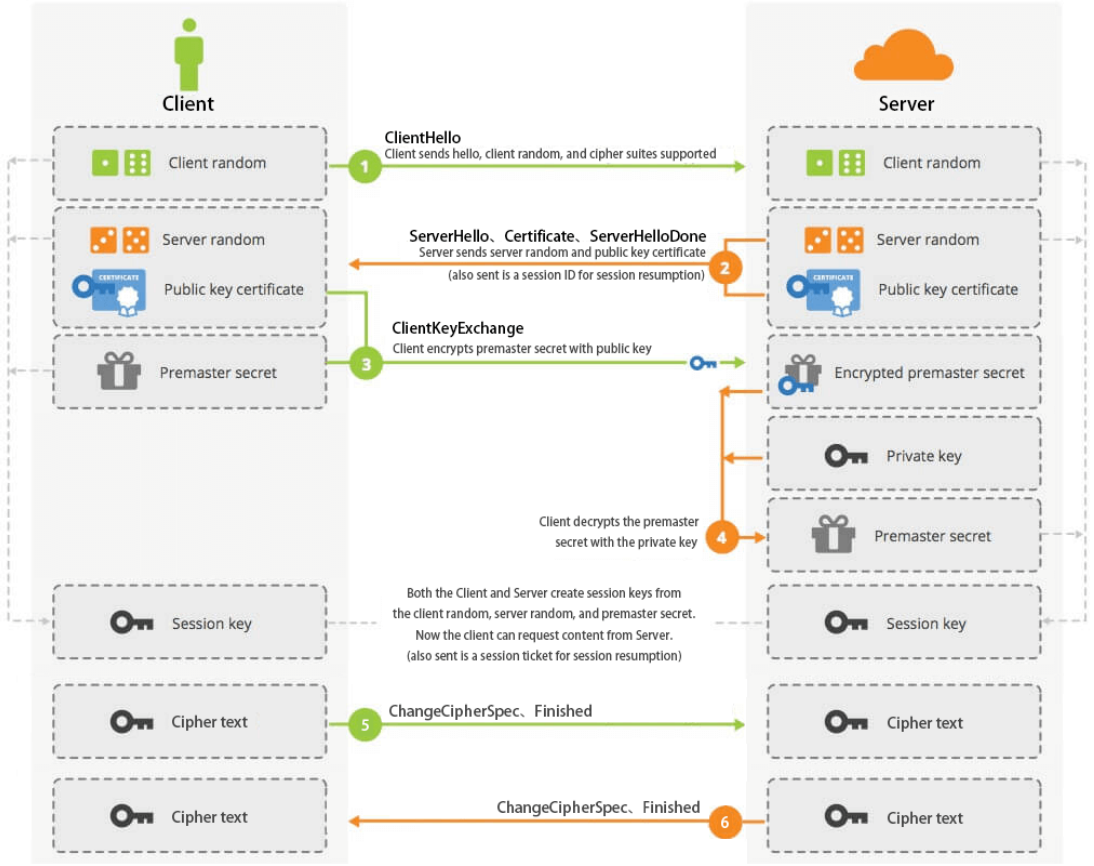
TCP connections commonly incur at least **one round-trip delay** of connection setup time before any application data can be sent, and TLS **adds two round trips** to this delay.



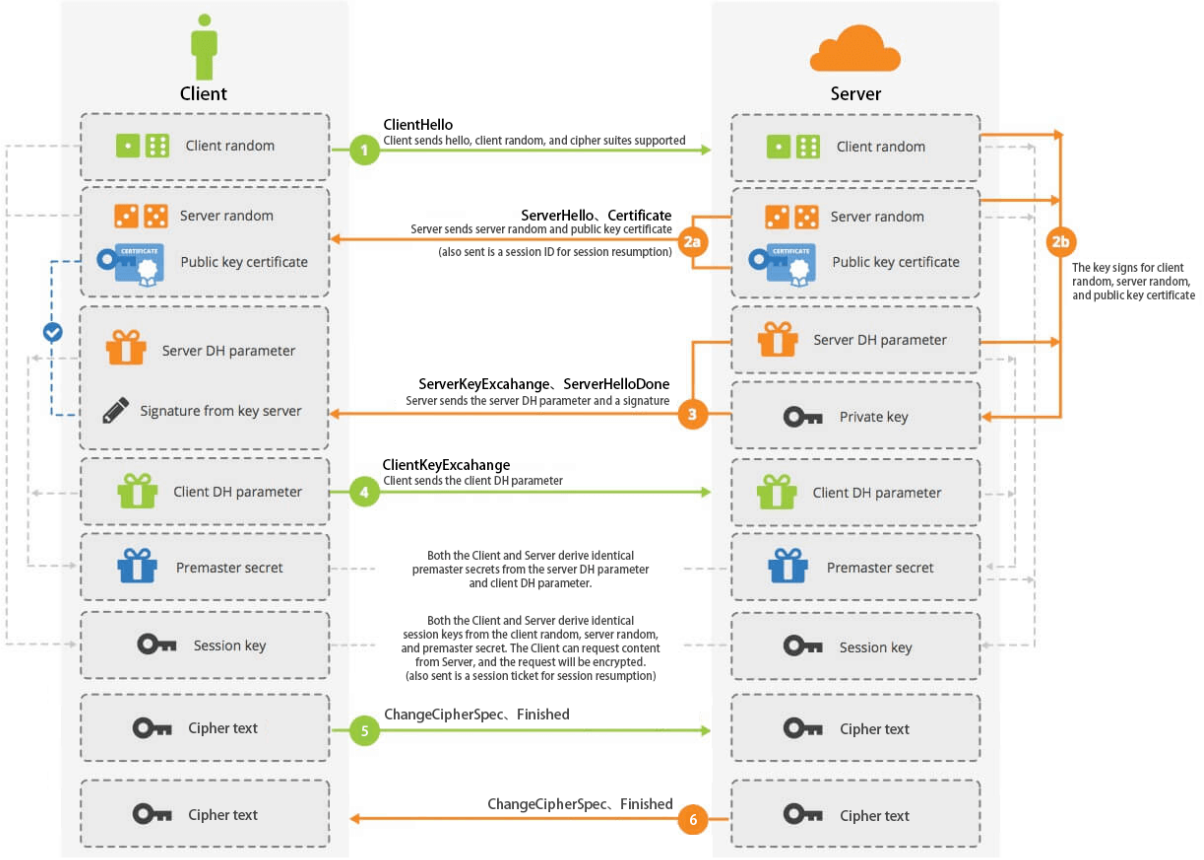
《TCP三次握手，四次挥手》
https://zhuanlan.zhihu.com/p/76874892

Handshake Delay

TLS Handshake RSA



TLS Handshake Diffie-Hellman



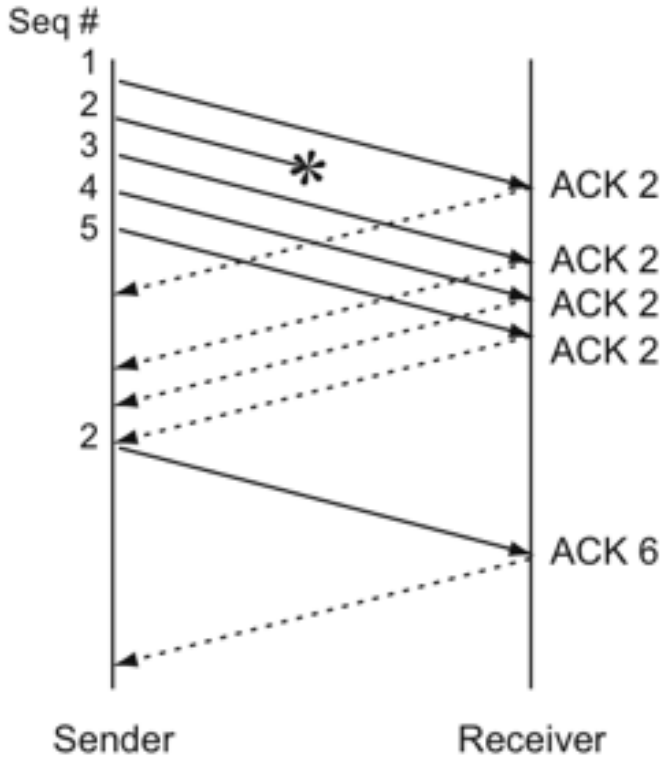
HTTPS 温故知新（三）—— 直观感受 TLS 握手流程(上)

https://halfrost.com/https_tls1-2_handshake/

Head-of-line Blocking Delay

TCP's bytestream abstraction, however, prevents applications from controlling the framing of their communications [12] and imposes a **"latency tax" on application frames** whose delivery must **wait for retransmissions of previously lost TCP segments.**

Sample of Fast Retransmit



浅谈TCP（1）：状态机与重传机制
https://monkeysayhi.github.io/2018/03/07/浅谈TCP（1）：状态机与重传机制/

QUIC’s Goal

QUIC is designed to meet several goals [59], including **deployability, security, and reduction in handshake and head-of-line blocking delays.**

The QUIC protocol combines its cryptographic and transport handshakes to minimize setup RTTs. It multiplexes multiple requests/responses over a single connection by providing each with its own stream, so that no response can be blocked by another.

DESIGN & IMPLEMENTATION

3

DESIGN & IMPLEMENTATION

- Connection Establishment
- Stream Multiplexing
- Authentication and Encryption
- Loss Recovery
- Flow Control
- Congestion Control
- NAT Rebinding and Connection Migration
- QUIC Discovery for HTTPS
- Open-Source Implementation

Connection Establishment

Annotations

- Inchoate client hello (CHLO)
- Reject (REJ)
- Server hello (SHLO)

An origin is identified by the set of URI scheme, hostname, and port number.

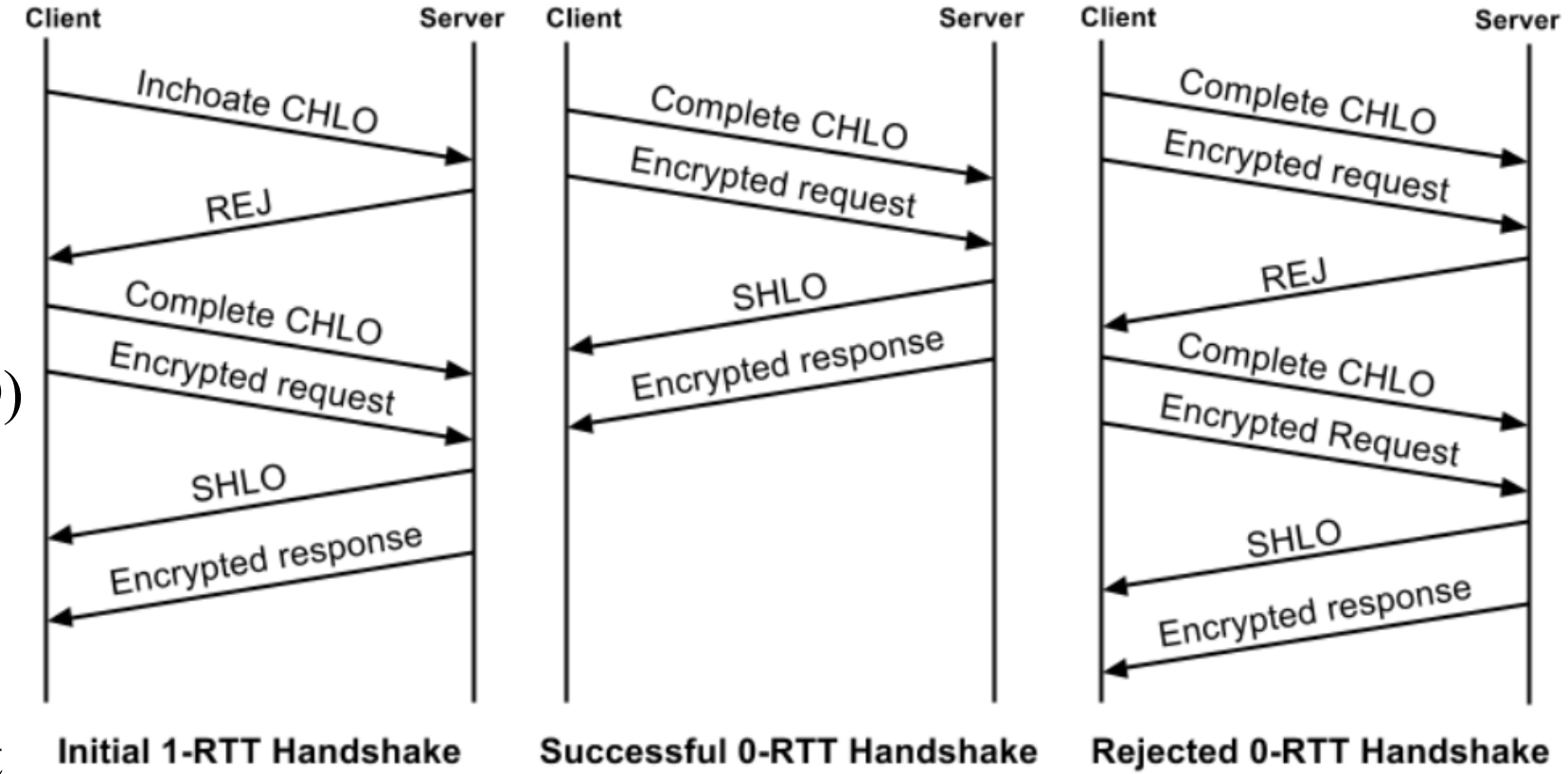


Figure 4: Timeline of QUIC's initial 1-RTT handshake, a subsequent successful 0-RTT handshake, and a failed 0-RTT handshake.

Connection Establishment

REJ Message from Server

- Server's long-term Diffie-Hellman public value
- A certificate chain authenticating the server
- A signature of the server config using the private key from the leaf certificate of the chain
- A source-address token: an authenticated-encryption block that contains the client's publicly visible IP address (as seen at the server) and a timestamp by the server. **The client sends this token back to the server in later handshakes, demonstrating ownership of its IP address.**

Connection Establishment

Annotations

- Inchoate client hello (CHLO)
- Reject (REJ)
- Server hello (SHLO)

An origin is identified by the set of URI scheme, hostname, and port number.

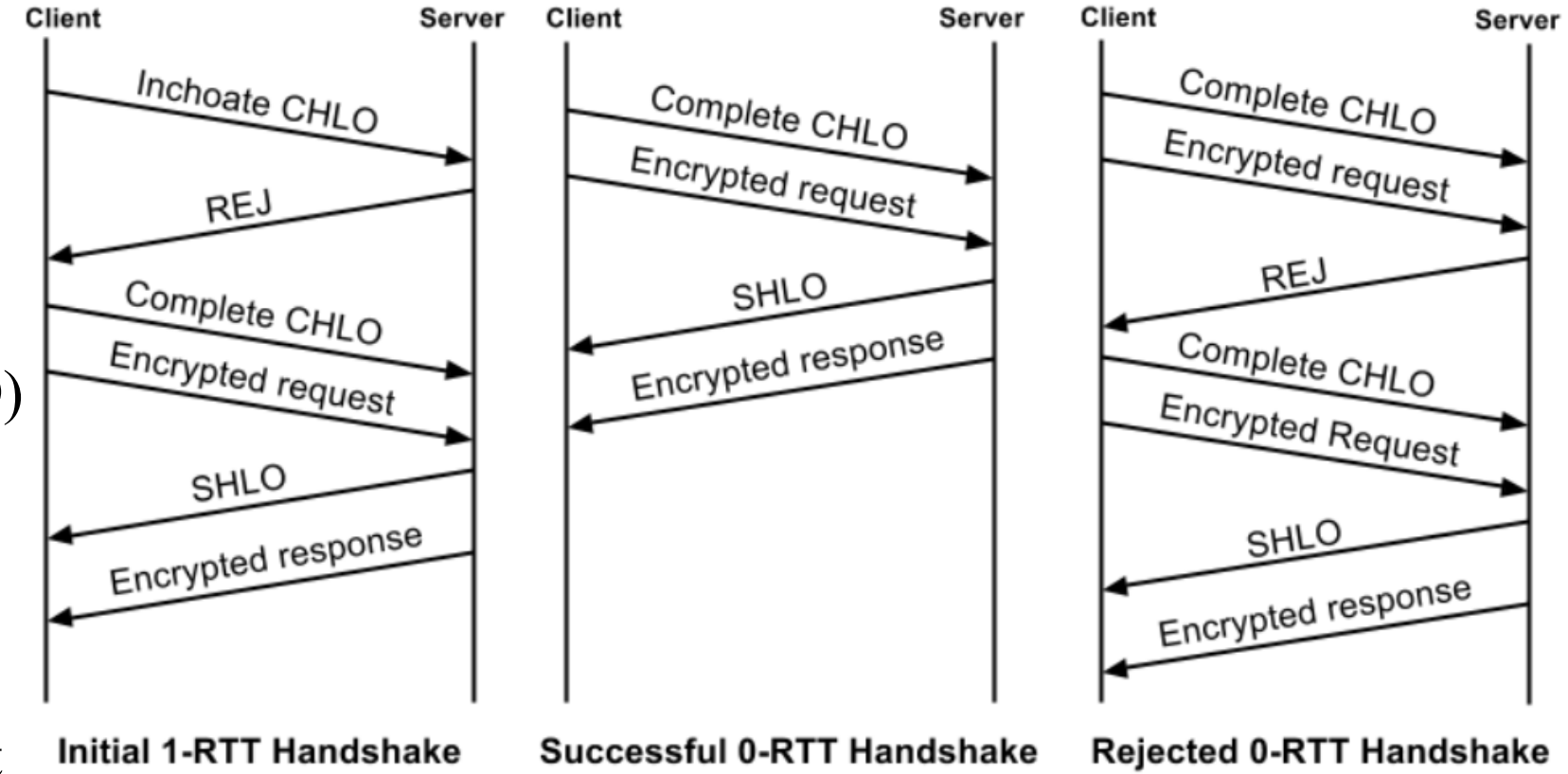


Figure 4: Timeline of QUIC's initial 1-RTT handshake, a subsequent successful 0-RTT handshake, and a failed 0-RTT handshake.

Connection Establishment

Version Negotiation

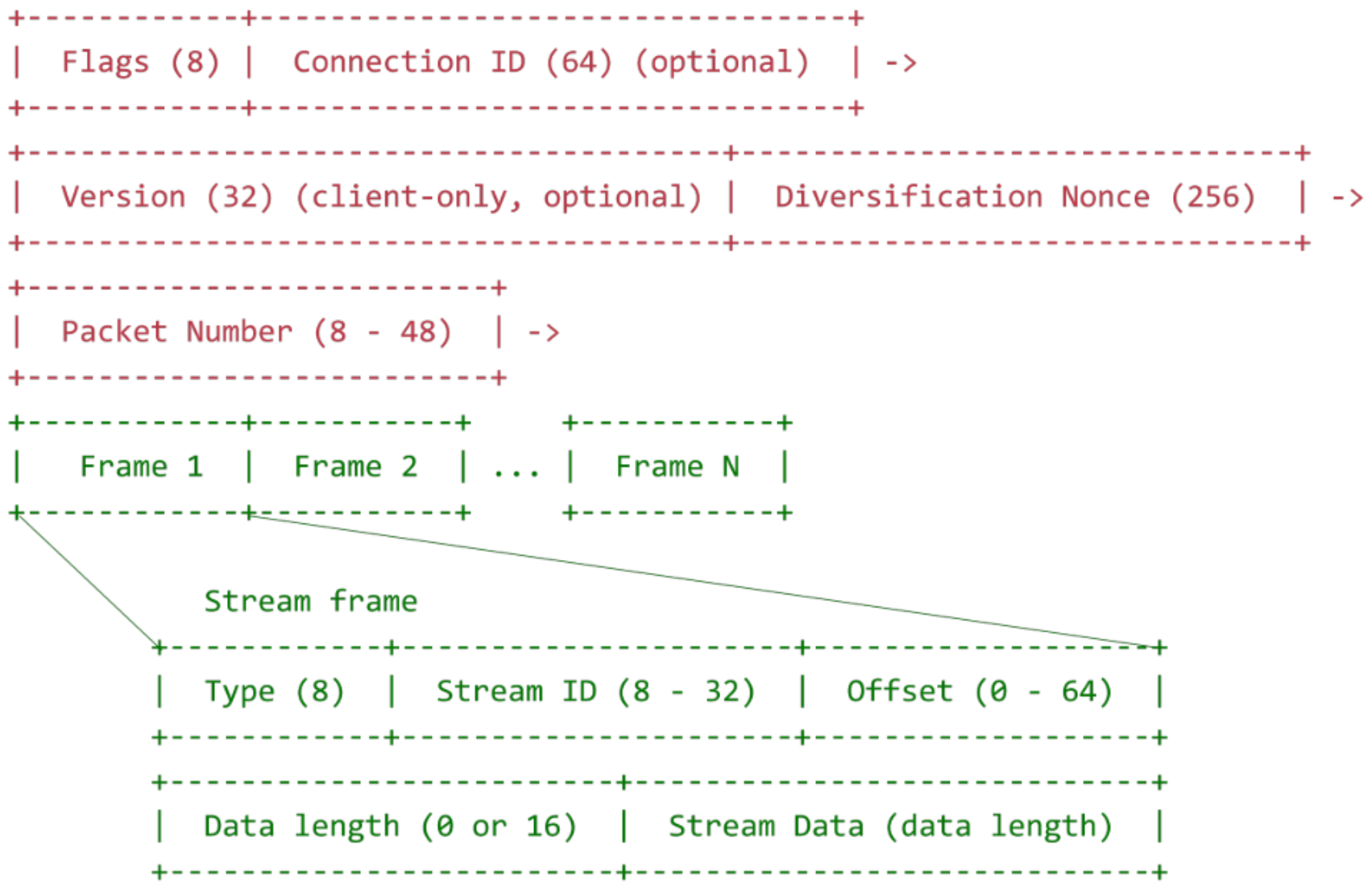
A QUIC client proposes a version to use for the connection **in the first packet of the connection** and encodes the rest of the handshake using the proposed version. If the server does not speak the client-chosen version, it forces version negotiation by **sending back a Version Negotiation packet** to the client carrying all of the server's supported versions, **causing a round trip of delay** before connection establishment.

This mechanism eliminates round-trip latency when the client's optimistically-chosen version is spoken by the server, and **incentivizes servers to not lag behind clients in deployment of newer versions.**

To prevent **downgrade attacks**, the initial version requested by the client and the list of versions supported by the server are **both fed into the key-derivation function at both the client and the server while generating the final keys.**

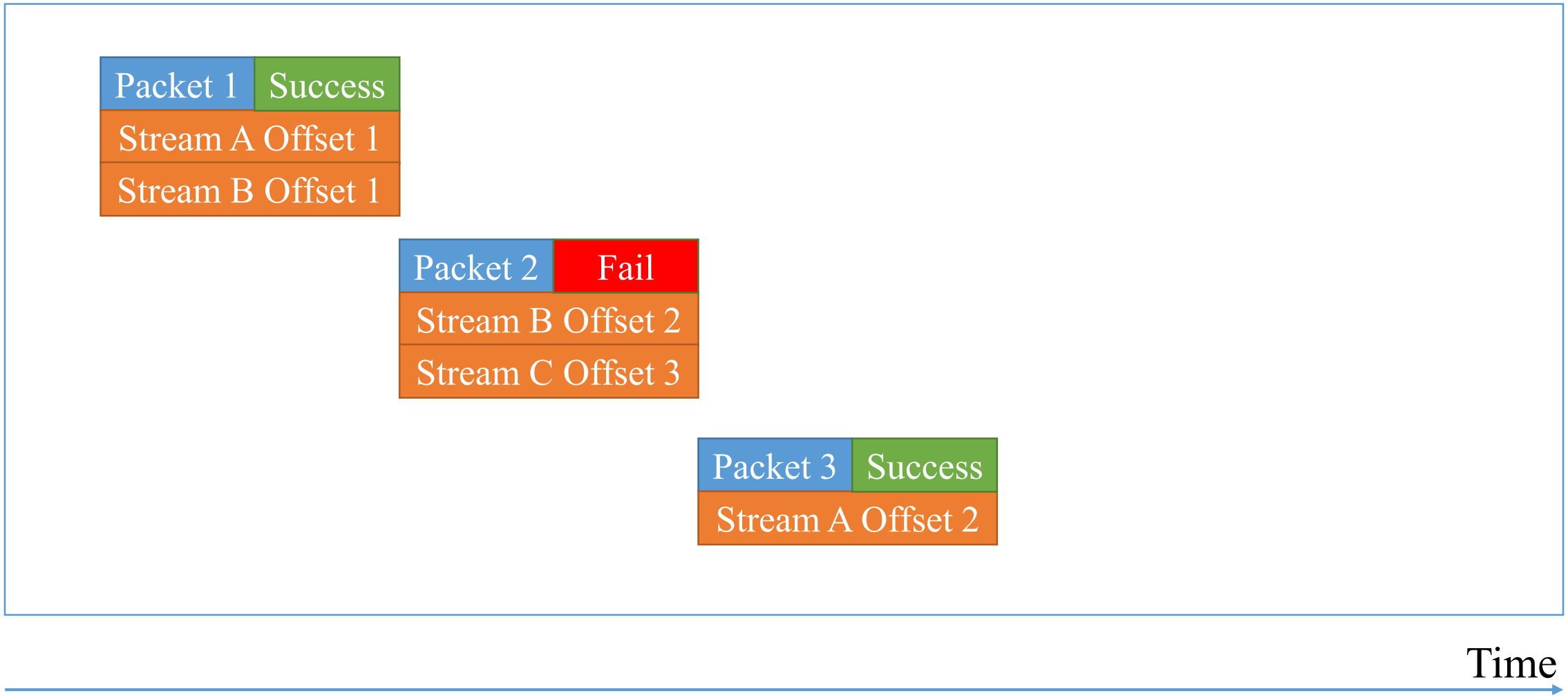
Stream Multiplexing

Applications commonly multiplex units of data within TCP's single bytestream abstraction. To avoid head-of-line blocking due to TCP's sequential delivery, QUIC supports multiple streams within a connection, ensuring that **a lost UDP packet only impacts those streams whose data was carried in that packet.**

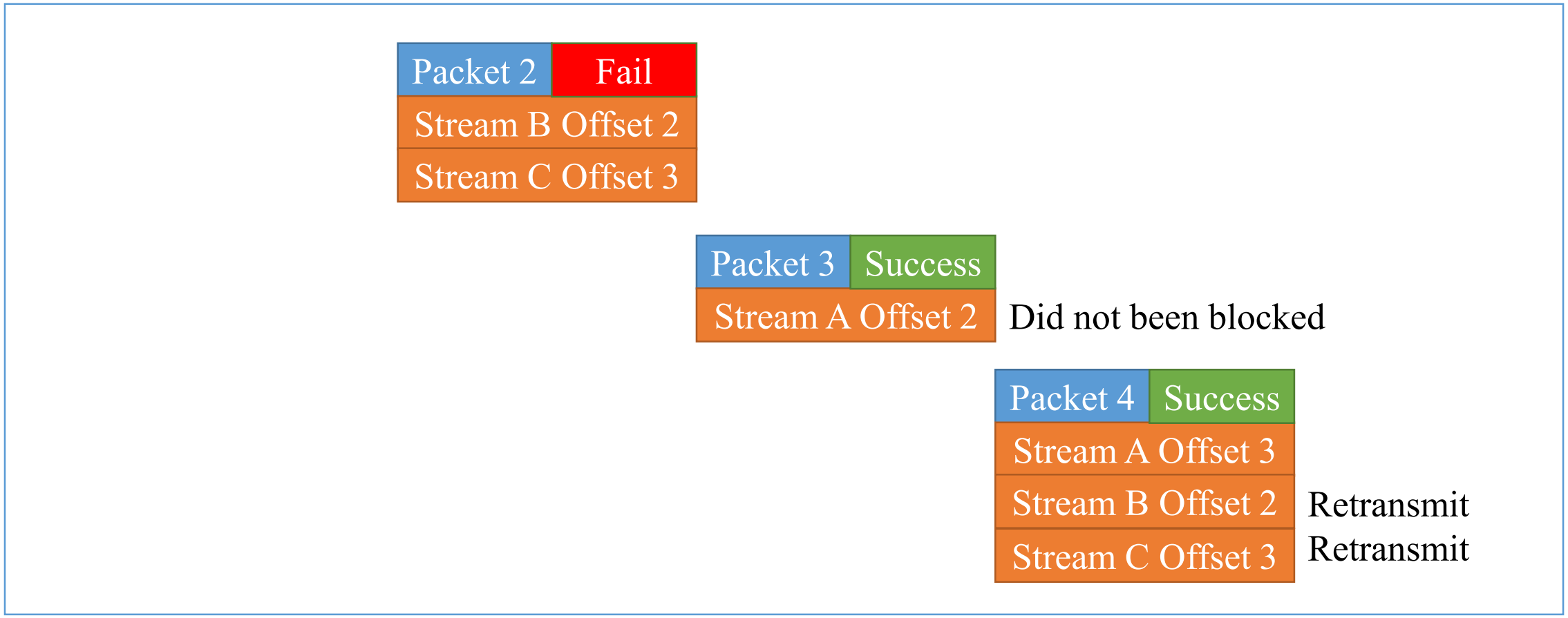


Padding Oracle Attack

Stream Multiplexing



Stream Multiplexing



Stream Multiplexing

Packet Types and Formats

QUIC has Special Packets and Regular Packets. There are two types of Special Packets: **Version Negotiation Packets and Public Reset Packets**, and regular packets containing frames.
All QUIC packets should be sized to fit within the path's MTU to avoid IP fragmentation. The **current QUIC implementation uses a 1350-byte maximum QUIC packet size for IPv6, 1370 for IPv4**. Both sizes are without IP and UDP overhead.

Frame Types and Formats

QUIC Frame Packets are populated by frames, which have a Frame Type byte, which itself has a type-dependent interpretation, followed by type-dependent frame header fields. **All frames are contained within single QUIC Packets and no frame can span across a QUIC Packet boundary.**

Authentication and Encryption

Cryptography - Diffie-Hellman

Suppose

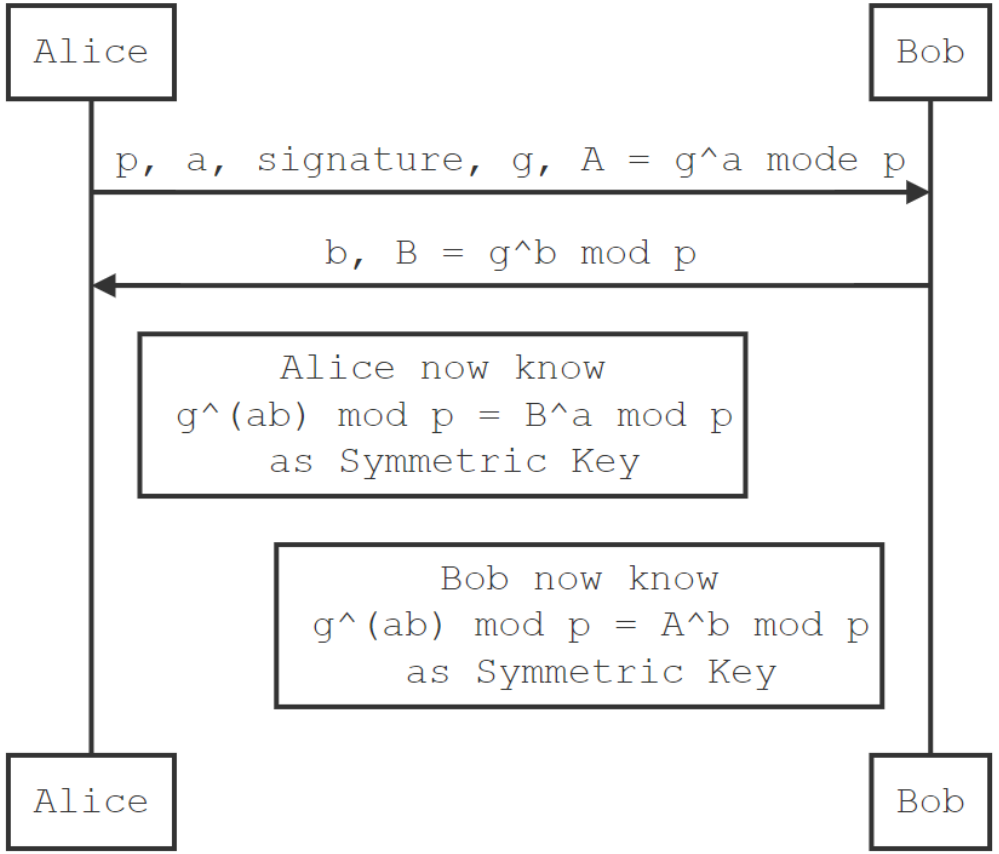
Client (Bob)

- *Private Key b*

Server (Alice)

- *A long term public value, large prime p*
- *Private Key a*
- *Signature by trusted third party if needed*
- *Public prime base g*

Goal: Symmetric Key g^(ab) mod p



Authentication and Encryption

Proof of Diffie-Hellman

Let:

$Prime\ p,\ g$

$Private\ Key\ a,\ b$

Proof:

$(g^a mod\ p)^b = (g^b mod\ p)^a = g^{ab} mod\ p$

Solution:

$\exists\ integer\ K\ let\ (g^a mod\ p)^b = (g^a + Kp)^b$

$\rightarrow (g^a + Kp)^b = [g^{ab} + C_b^1 g^{a-1} pK + C_b^2 g^{a-2} (pK)^2 + \ldots + (pK)^a]$

$\rightarrow (g^a + Kp)^b = g^{ab} mod\ p$

$\rightarrow (g^a mod\ p)^b = g^{ab} mod\ p$

Authentication and Encryption

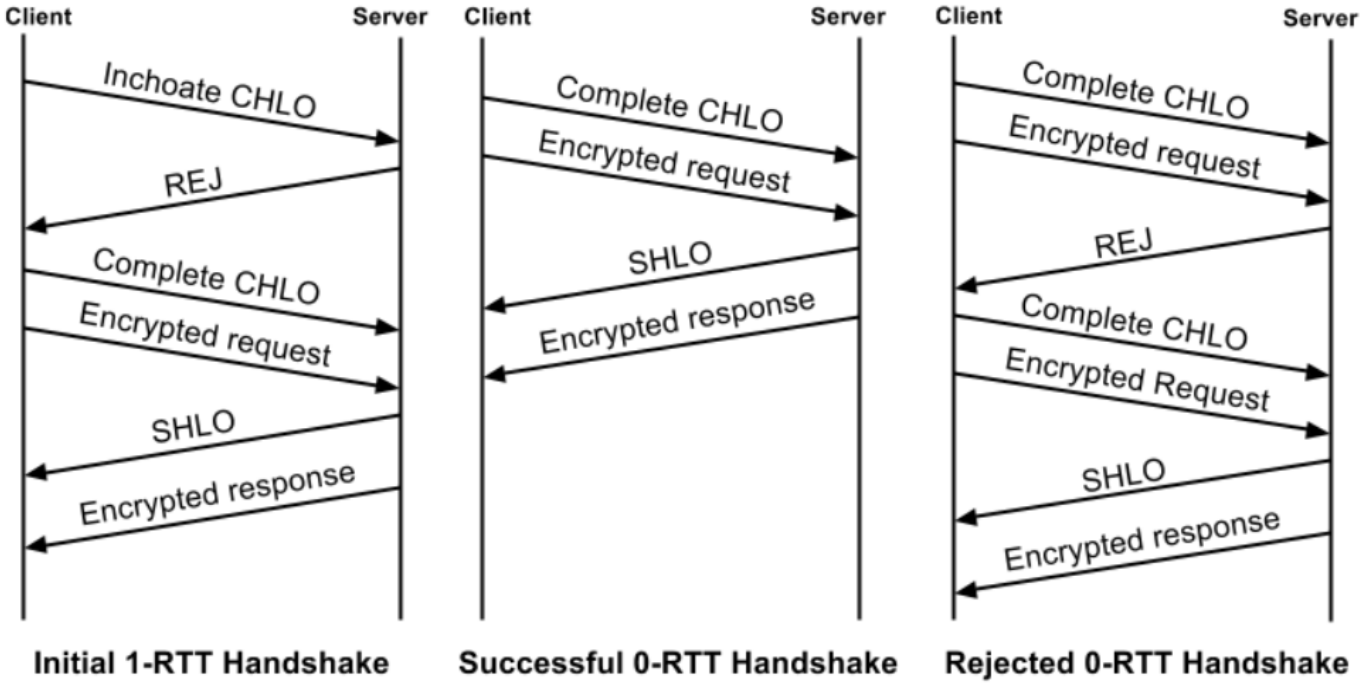
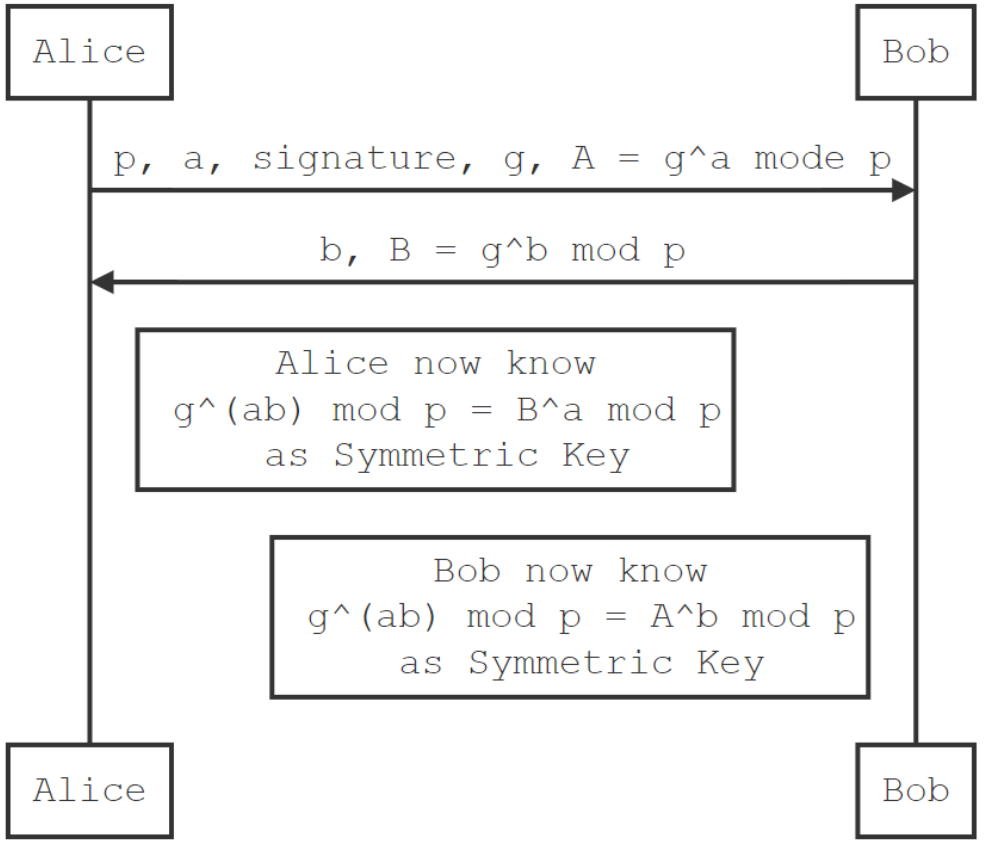


Figure 4: Timeline of QUIC's initial 1-RTT handshake, a subsequent successful 0-RTT handshake, and a failed 0-RTT handshake.

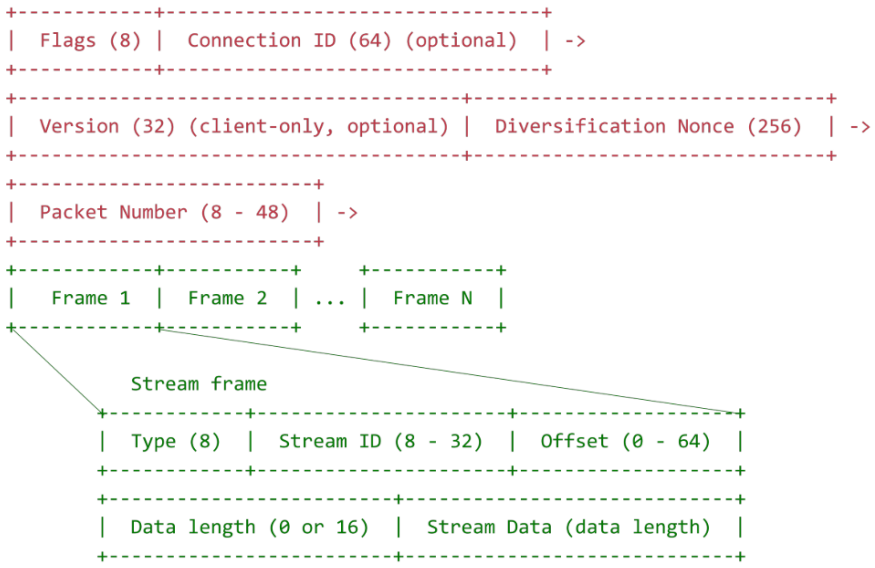


Authentication and Encryption

With the exception of **a few early handshake packets and reset packets**, QUIC packets are fully authenticated and mostly encrypted.

Upon sending an SHLO message, the server immediately switches to sending packets encrypted with the forward-secure keys. Upon receiving the SHLO message, the client switches to sending packets encrypted with **the forward-secure keys**.

Any information sent in unencrypted handshake packets, such as in the Version Negotiation packet, **is included in the derivation of the final connection keys.**



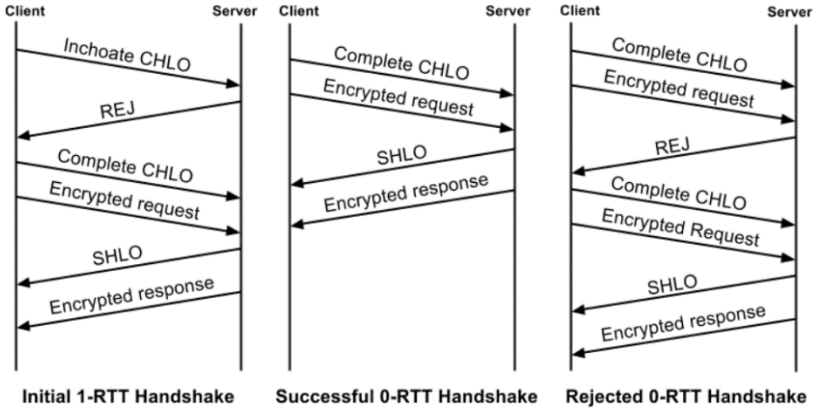
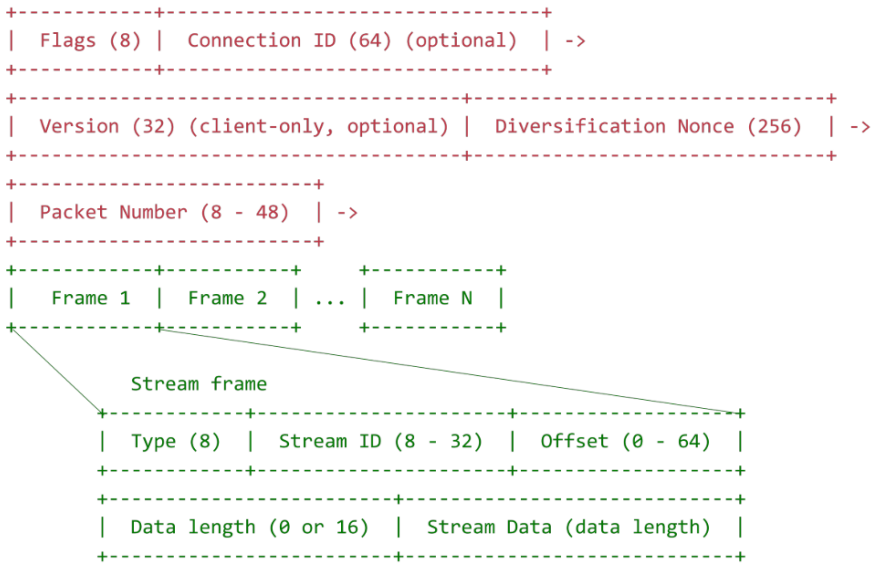


Figure 4: Timeline of QUIC's initial 1-RTT handshake, a subsequent successful 0-RTT handshake, and a failed 0-RTT handshake.

Authentication and Encryption

Forward Secrecy

In cryptography, **forward secrecy** (**FS**), also known as **perfect forward secrecy** (**PFS**), is a feature of specific key agreement protocols that gives assurances that session keys will not be compromised even if the private key of the server is compromised.
Forward secrecy protects past sessions **against future compromises of secret keys.**



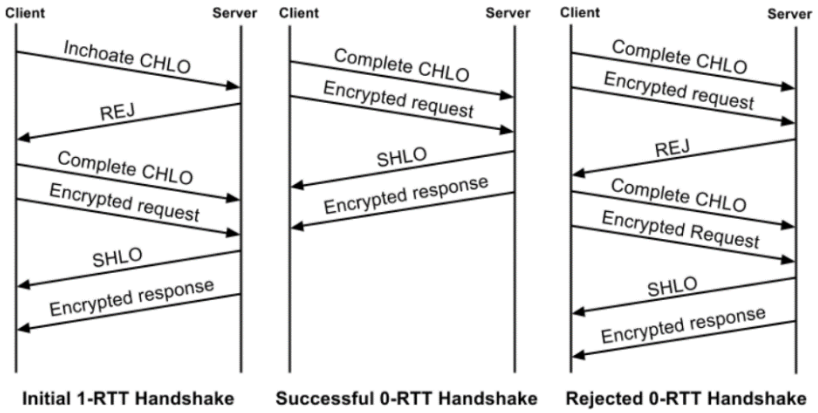


Figure 4: Timeline of QUIC's initial 1-RTT handshake, a subsequent successful 0-RTT handshake, and a failed 0-RTT handshake.

Authentication and Encryption

Forward Secrecy

REJ: containing server's long-term Diffie-Hellman public value
complete CHLO: containing the client's ephemeral Diffie-Hellman public value.
SHLO: This message is encrypted using the initial keys, and contains the server's ephemeral Diffie-Hellman public value.

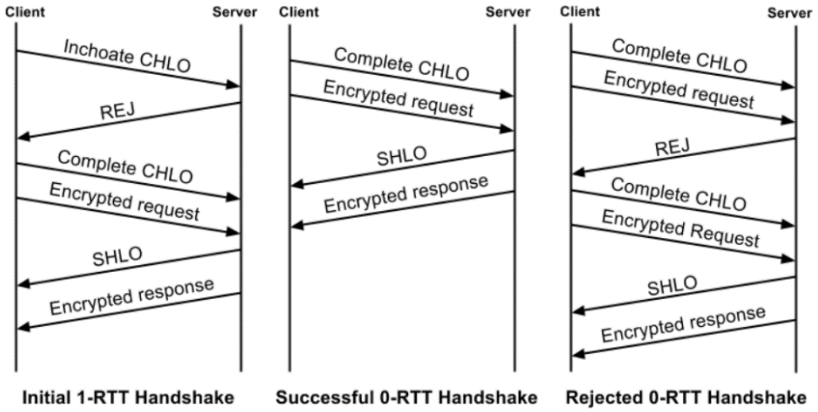


Figure 4: Timeline of QUIC's initial 1-RTT handshake, a subsequent successful 0-RTT handshake, and a failed 0-RTT handshake.

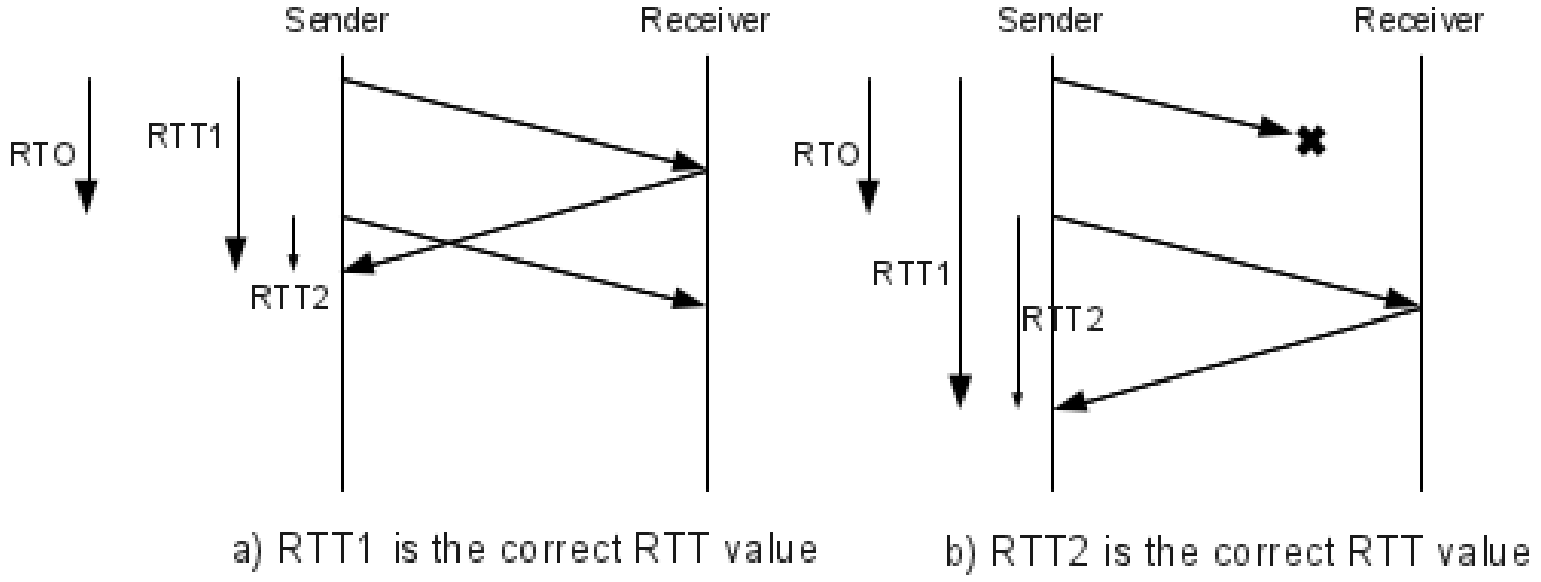
QUIC's cryptography therefore provides two levels of secrecy
Initial client data is encrypted using initial keys.
Subsequent client data and all server data are encrypted using forward-secure keys.

Loss Recovery

Each QUIC packet carries **a new packet number**, **including** those carrying retransmitted data. This design obviates the need for a separate mechanism to **distinguish the ACK of a retransmission from that of an original transmission**, thus avoiding TCP's retransmission ambiguity problem. Stream offsets in stream frames are used for delivery ordering, separating the two functions that TCP conflates. The packet number represents an explicit time-ordering, which enables simpler and more accurate loss detection than in TCP.

RTT Sampling Ambiguity

A timeout occurs before an ACK is received, and PKT1 is retransmitted. The ACK for the first PKT1 arrives a bit later and the source measures a wrong value for the RTT.



TCP RTT Estimation, Fast Retransmit, Fast Recovery, Rupesh R. Mehta
https://www.cse.iitb.ac.in/~br/webpage/courses/cs625-fall2003/lec-notes/lec-notes11-1.html

Calculating TCP RTO...
http://sgros.blogspot.com/2012/02/calculating-tcp-rto.html

Loss Recovery

Stream offsets in stream frames are used for **delivery ordering**, separating the two functions that TCP conflates. The packet number represents an **explicit time-ordering**, which enables simpler and more accurate loss detection than in TCP.

QUIC's acknowledgments support up to 256 ACK blocks, making QUIC more resilient to reordering and loss than TCP with SACK [46].

Advantages

Accurate RTT estimation can also aid delay-sensing congestion controllers such as BBR [10] and PCC [16].

These differences between QUIC and TCP allowed us to build simpler and more effective mechanisms for QUIC.

Flow Control

Similar to HTTP/2 [8], QUIC employs **credit-based flow-control.**

A QUIC receiver advertises the absolute byte offset within each stream up to which the receiver is willing to receive data.
As data is sent, received, and delivered on a particular stream, the receiver periodically sends window update frames that increase the advertised offset limit for that stream, allowing the peer to send more data on that stream. Connection-level flow control works in the same way as stream-level flow control, but the bytes delivered and the highest received offset are **aggregated across all streams.**

Congestion Control

The QUIC protocol does not rely on a specific congestion control algorithm and our implementation has a **pluggable interface** to allow experimentation.

Pluginizing QUIC

NAT Rebinding and Connection Migration

QUIC connections are identified by a **64-bit Connection ID**.

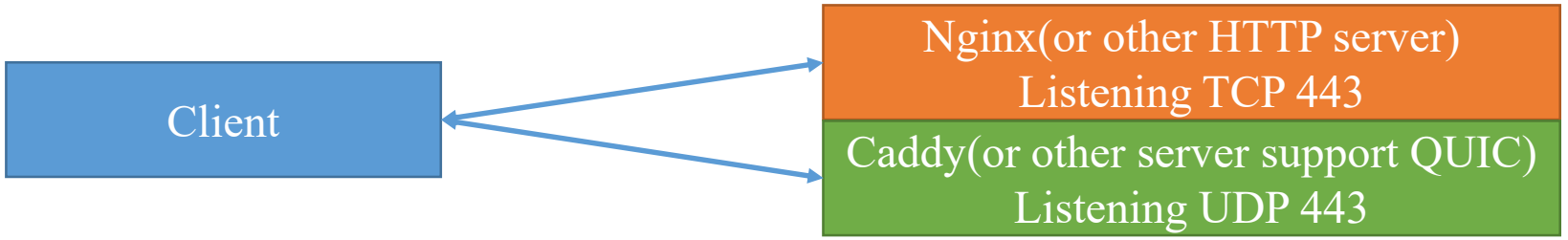
QUIC's Connection ID enables connections to survive changes to the client's IP and port. Such changes can be caused by NAT timeout and rebinding (which tend to be more aggressive for UDP than for TCP [27]) or by the client changing network connectivity to a new IP address.

While QUIC endpoints simply elide the problem of NAT rebinding by using the Connection ID to identify connections, client-initiated connection migration is a work in progress with limited deployment at ~~this~~(that) point.

QUIC Discovery for HTTPS

A client does not know a priori whether a given server speaks QUIC. When our client makes an HTTP request to an origin for the first time, it sends the request over TLS/TCP. Our servers **advertise QUIC support by including an "Alt-Svc" header in their HTTP responses** [48]. This header tells a client that connections to the origin may be attempted using QUIC. The client can now attempt to use QUIC in subsequent requests to the same origin.

On a subsequent HTTP request to the same origin, the client **races a QUIC and a TLS/TCP connection**, but prefers the QUIC connection by delaying connecting via TLS/TCP by up to 300 ms. Whichever protocol successfully establishes a connection first ends up getting used for that request. If QUIC is blocked on the path, or if the QUIC handshake packet is larger than the path's MTU, then the QUIC handshake fails, and the client uses the fallback TLS/TCP connection.

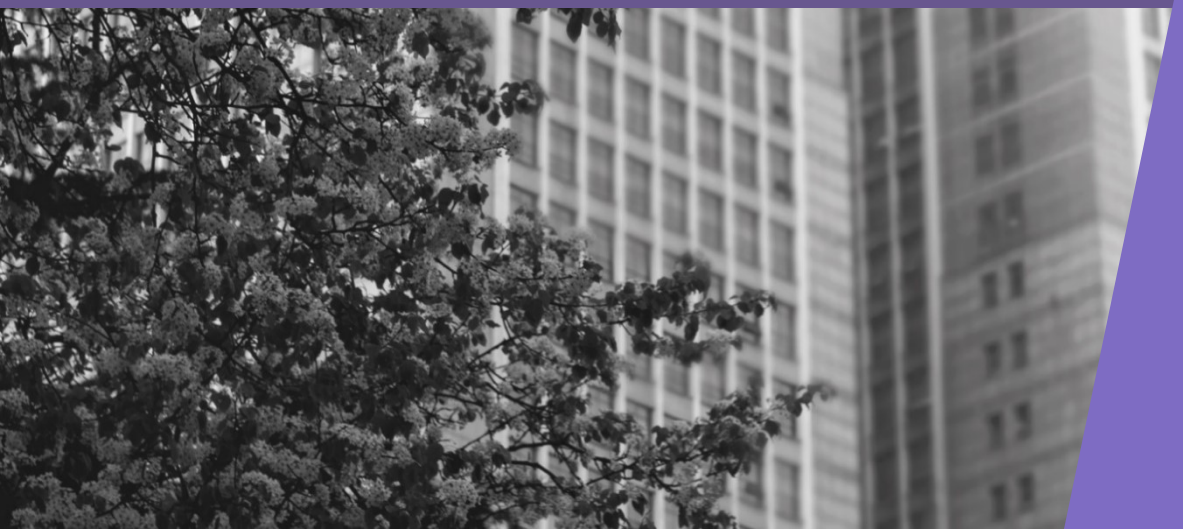


Open-Source Implementation

Our implementation of QUIC is available as part of the open-source Chromium project [1]. This implementation is shared code, used by Chrome and other clients such as YouTube, and also by Google servers albeit with additional Google-internal hooks and protections. The source code is in C++, and includes substantial unit and end-to-end testing. The implementation includes a test server and a test client which can be used for experimentation, but are not tuned for production-level performance.

EXPERIMENTS & EXPERIENCES

4



EXPERIMENTS & EXPERIENCES

Experiments

- Experimentation Framework
- QUIC Performance
- Performance By Region
- Server CPU Utilization
- Performance Limitations

Experiences

- Packet Size Considerations
- UDP Blockage and Throttling
- Forward Error Correction
- User-space Development
- Experiences with Middleboxes

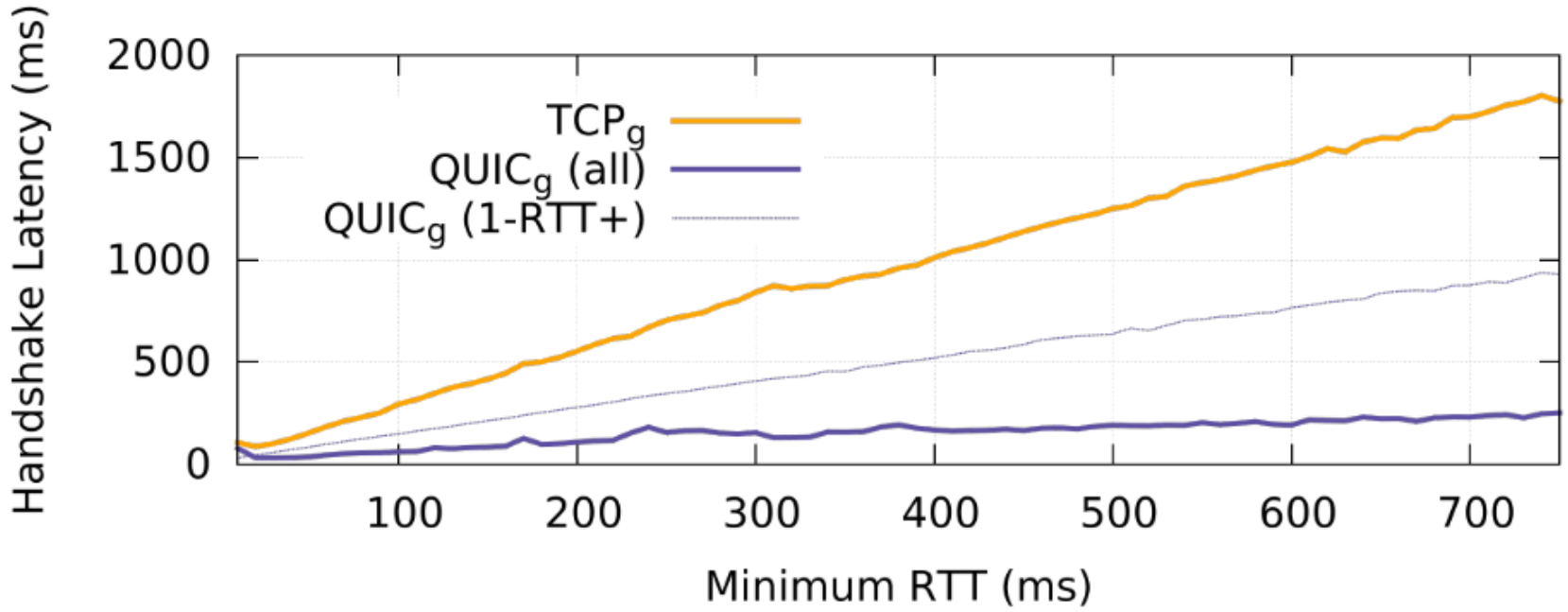
Experimentation Framework

Our development of the QUIC protocol relies heavily on continual Internet-scale experimentation to examine the value of various features and to tune parameters. In this section we describe the experimentation frameworks in Chrome and our server fleet, which allow us to experiment safely with QUIC.

We drove QUIC experimentation by **implementing it in Chrome**, which has a strong experimentation and analysis framework that allows new features to be A/B tested and evaluated before full launch. **Chrome's experimentation framework pseudo-randomly assigns clients to experiments and exports a wide range of metrics, from HTTP error rates to transport handshake latency.** Clients that are opted into statistics gathering report their statistics along with a list of their assigned experiments, which subsequently enables us to slice metrics by experiment. This framework also allows us to rapidly disable any experiment, thus protecting users from problematic experiments.

QUIC Performance

Handshake Latency is the amount of time taken to establish a secure transport connection. In TLS/TCP, this includes the time for **both the TCP and TLS handshakes to complete.** We measured handshake latency at the server as the time **from receiving the first TCP SYN or QUIC client hello packet** to **the point at which the handshake is considered complete**. In the case of a QUIC 0-RTT handshake, latency is measured as 0 ms. Figure shows the impact of QUIC's 0-RTT and 1-RTT handshakes on handshake latency.



QUIC Performance

Search Latency is the delay between when **a user enters a search term** and when **all the search-result content is generated and delivered to the client by Google Search, including** all corresponding images and embedded content. On average, an individual search performed by a user results in a total response load of 100 KB for desktop searches and 40 KB for mobile searches. As a metric, Search Latency represents delivery latency for small, delay-sensitive, dynamically-generated payloads.

Video Latency for a video playback is measured as **the time between when a user hits "play" on a video to when the video starts playing.** To ensure smooth playbacks, video players typically buffer a couple seconds of video before playing the first frame. The amount of data the player loads depends on the bitrate of the playback.

Video Rebuffer Rate, or simply ***Rebuffer Rate*** is the percentage of time that a video pauses during a playback to rebuffer data normalized by video watch time, where video watch time includes time spent rebuffering. In other words, Rebuffer Rate is computed as (Rebuffer Time) / (Rebuffer Time + Video Play Time).

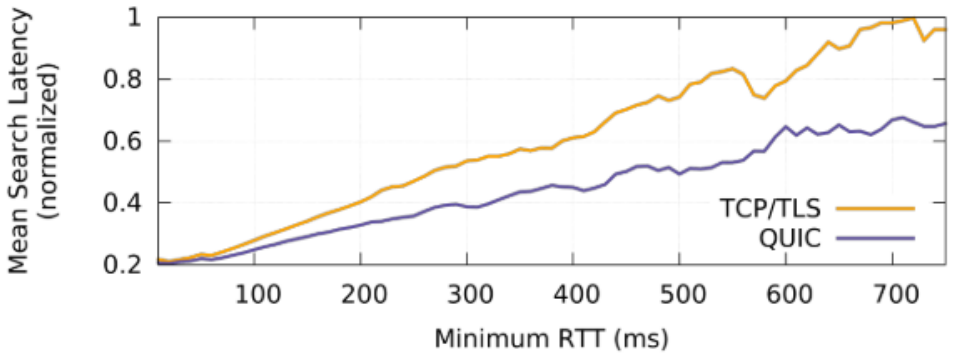
QUIC Performance

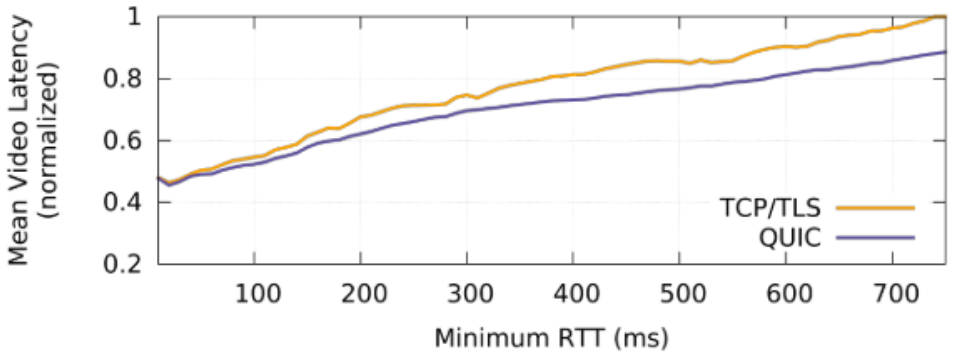
		% latency reduction by percentile						
		Lower latency				Higher latency		
	Mean	1%	5%	10%	50%	90%	95%	99%
Search								
Desktop	8.0	0.4	1.3	1.4	1.5	5.8	10.3	16.7
Mobile	3.6	-0.6	-0.3	0.3	0.5	4.5	8.8	14.3
Video								
Desktop	8.0	1.2	3.1	3.3	4.6	8.4	9.0	10.6
Mobile	5.3	0.0	0.6	0.5	1.2	4.4	5.8	7.5

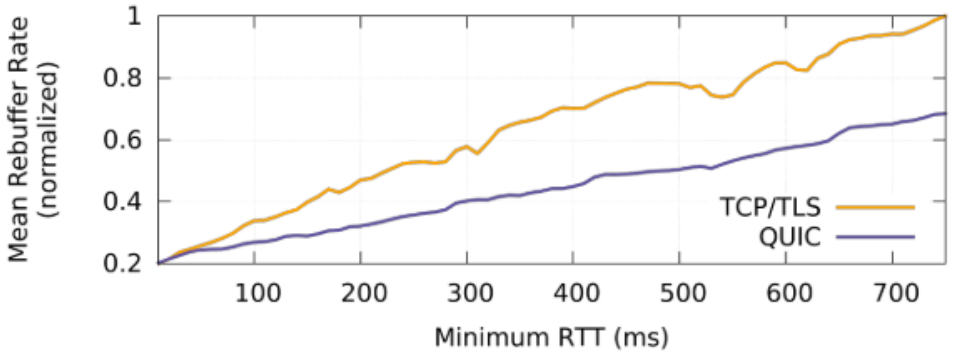
Table 1: Percent reduction in global Search and Video Latency for users in $QUIC_g$, at the mean and at specific percentiles. A 16.7% reduction at the 99th percentile indicates that the 99th percentile latency for $QUIC_g$ is 16.7% lower than the 99th percentile latency for TCP_g.

		% rebuffer rate reduction by percentile				
		Fewer rebuffers			More rebuffers	
	Mean	< 93%	93%	94 %	95%	99%
Desktop	18.0	*	100.0	70.4	60.0	18.5
Mobile	15.3	*	*	100.0	52.7	8.7

Table 2: Percent reduction in global Video Rebuffer Rate for users in $QUIC_g$ at the mean and at specific percentiles. An 18.5% reduction at the 99th percentile indicates that the 99th percentile rebuffer rate for $QUIC_g$ is 18.5% lower than the 99th percentile rate for TCP_g. An * indicates that neither $QUIC_g$ nor TCP_g have rebuffers at that percentile.







Performance By Region

Differences in access-network quality and distance from Google servers result in RTT and retransmission rate variations for **different geographical regions**. We now look at QUIC's impact on Search Latency and on Video Rebuffer Rate in select countries, chosen to span a wide range of network conditions.

Table 3 show how QUIC's performance impact varies by country.

In South Korea, which has the **lowest average RTT and the lowest network loss**, QUIC's performance is closer to that of TCP. Network conditions in the United States are **more typical of the global average**, and QUIC shows greater improvements in the USA than in South Korea. India, which **has the highest average RTT and retransmission rate**, shows the highest benefits across the board. QUIC's performance benefits over TLS/TCP are thus not uniformly distributed across geography or network quality: benefits are greater in networks and regions that have higher average RTT and higher network loss.

			% Reduction in Search Latency		% Reduction in Rebuffer Rate	
Country	Mean Min RTT (ms)	Mean TCP Rtx %	Desktop	Mobile	Desktop	Mobile
South Korea	38	1	1.3	1.1	0.0	10.1
USA	50	2	3.4	2.0	4.1	12.9
India	188	8	13.2	5.5	22.1	20.2

Table 3: Network characteristics of selected countries and the changes to mean Search Latency and mean Video Rebuffer Rate for users in $QUIC_g$.

Server CPU Utilization

The QUIC implementation was initially written with a focus on rapid feature development and ease of debugging, not CPU efficiency. When we started measuring the cost of serving YouTube traffic over QUIC, we found that QUIC's server CPU-utilization was about **3.5 times higher** than TLS/TCP.
The three major sources of QUIC's CPU cost were: **cryptography, sending and receiving of UDP packets, and maintaining internal QUIC state.**

To reduce cryptographic costs, we employed a hand-optimized version of the **ChaCha20 cipher favored by mobile clients.**
To reduce packet receive costs, we used asynchronous packet reception from the kernel via a memory-mapped application ring buffer (Linux's PACKET_RX_RING).
Finally, to reduce the cost of maintaining state, we rewrote critical paths and data-structures to be **more cache-efficient.**

With these optimizations, we decreased the CPU cost of serving web traffic over QUIC to **approximately twice that of TLS/TCP**, which has allowed us to increase the levels of QUIC traffic we serve.

Performance Limitations

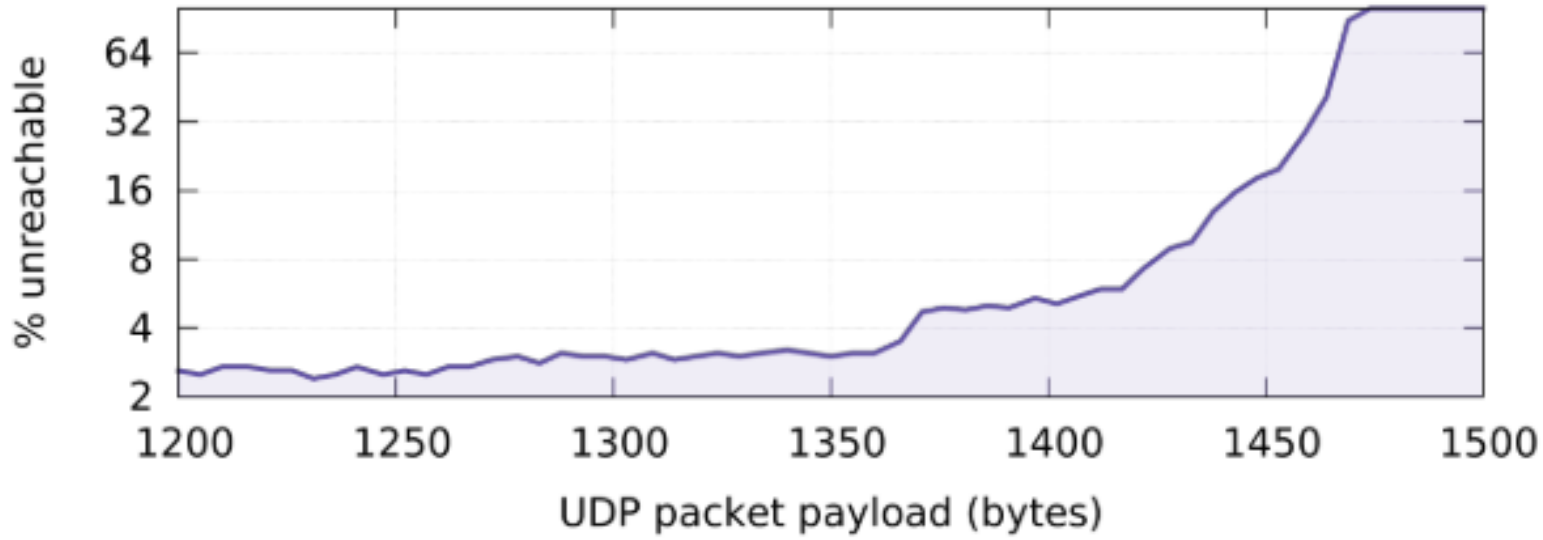
Pre-warmed connections: When applications hide handshake latency by performing handshakes proactively, these applications receive no measurable benefit from QUIC's 0-RTT handshake.

High bandwidth, low-delay, low-loss networks: The use of QUIC on networks with plentiful bandwidth, low delay, and low loss rate, shows little gain and occasionally negative performance impact. When used over a very high-bandwidth (over 100 Mbps) and/or very low RTT connection (a few milliseconds), QUIC may perform worse than TCP.

Mobile devices: QUIC's gains for mobile users are generally more modest than gains for desktop users.

Packet Size Considerations

We tested a range of possible UDP payload sizes, **from 1200 bytes up to 1500 bytes, in 5 byte increments**. For each packet size, **approximately 25,000 instances** of Chrome would attempt to send UDP packets of that size to an echo server on our network and wait for a response. If at least one response was received, this trial counted as a reachability success, otherwise it was considered to be a failure.



UDP Blockage and Throttling

QUIC is successfully used for 95.3% of video clients attempting to use QUIC. 4.4% of clients are unable to use QUIC, meaning that **QUIC or UDP is blocked** or **the path's MTU is too small**. Manual inspection showed that these users are commonly found in corporate networks, and are likely **behind enterprise firewalls**. **We have not seen an entire ISP blocking QUIC or UDP.**

The remaining 0.3% of users are in networks that seem to rate limit QUIC and/or UDP traffic. We detect rate limiting as substantially elevated packet loss rate and decreased bandwidth **at peak times of day, when traffic is high.**

Forward Error Correction

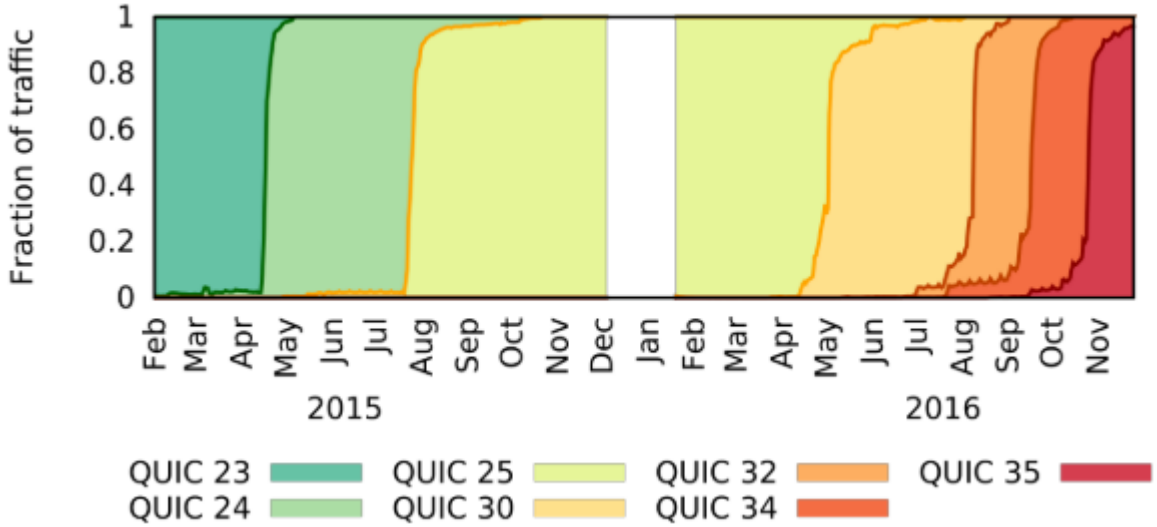
We experimented with various packet protection policies—protecting only HTTP headers, protecting all data, sending an FEC packet only on quiescence—and found similar outcomes. While retransmission rates decreased measurably, **FEC had statistically insignificant impact** on Search Latency and increased both Video Latency and Video Rebuffer Rate for video playbacks.

In addition to benefits that were **not compelling**, implementing FEC **introduced a fair amount of code complexity.** Consequently, we removed support for XOR-based FEC from QUIC in early 2016.

User-space Development

Development practices directly influence robustness of deployed code. In keeping with modern software development practices, we **relied heavily on extensive unit and end-to-end testing**. We used a network simulator built into the QUIC code to perform fine-grained congestion control testing. Such facilities, which are often limited in kernel development environments, **frequently caught significant bugs prior to deployment and live experimentation.**

An added benefit of user-space development is that a user-space application is **not as memory-constrained as the kernel**, is **not limited by the kernel API**, and can freely interact with other systems in the server infrastructure. This allows for **extensive logging and integration with a rich server logging infrastructure**, which is invaluable for debugging.



Experiences with Middleboxes

In October 2016, we introduced a 1-bit change to the public flags field of the QUIC packet header. **This change resulted in pathological packet loss for users behind one brand of firewall that supported explicit blocking of QUIC traffic.** In previous versions of QUIC, this firewall correctly blocked all QUIC packets, causing clients to fall back to TCP. The firewall used the QUIC flags field to identify QUIC packets, and the 1-bit change in the flags field confounded this detection logic, causing the firewall **to allow initial packets through but blocking subsequent packets**. As a result, clients that were previously using TCP (since QUIC was previously successfully blocked) were now starting to use QUIC and then hitting **a packet black-hole**. The problem was fixed by reverting the flags change across our clients.

Experiences with Middleboxes

While we were able to isolate the problem to one vendor in this instance, our process of reaching out to them does not scale to all middleboxes and vendors. **We do not know if other bits exposed by QUIC have been ossified by this or other middleboxes, and we do not have a method to answer this question at Internet scale.** We did learn that middlebox vendors are reactive. When traffic patterns change, they build responses to these observed changes. This pattern of behavior exposes a **"deployment impossibility cycle"** however, since deploying a protocol change widely requires it to work through a huge range of middleboxes, but middleboxes only change behavior in response to wide deployment of the change. This experience reinforces the premise on which QUIC was designed: when deploying end-to-end changes, encryption is the only means available to ensure that bits that ought not be used by a middlebox are in fact not used by one.

THANKS