The QUIC Transport Protocol
- Design and Internet-Scale Deployment

Chen, Zixuan

这里填写大标题

OUTLINE

0

Outline

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.

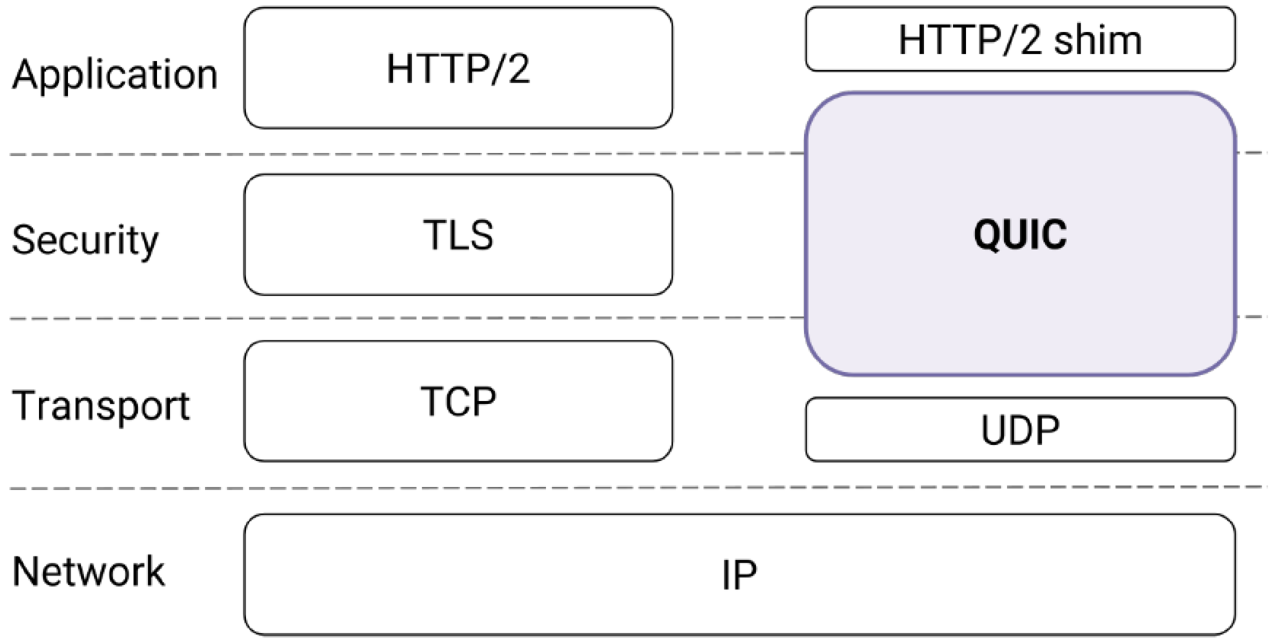


Figure 1: QUIC in the traditional HTTPS stack.

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].

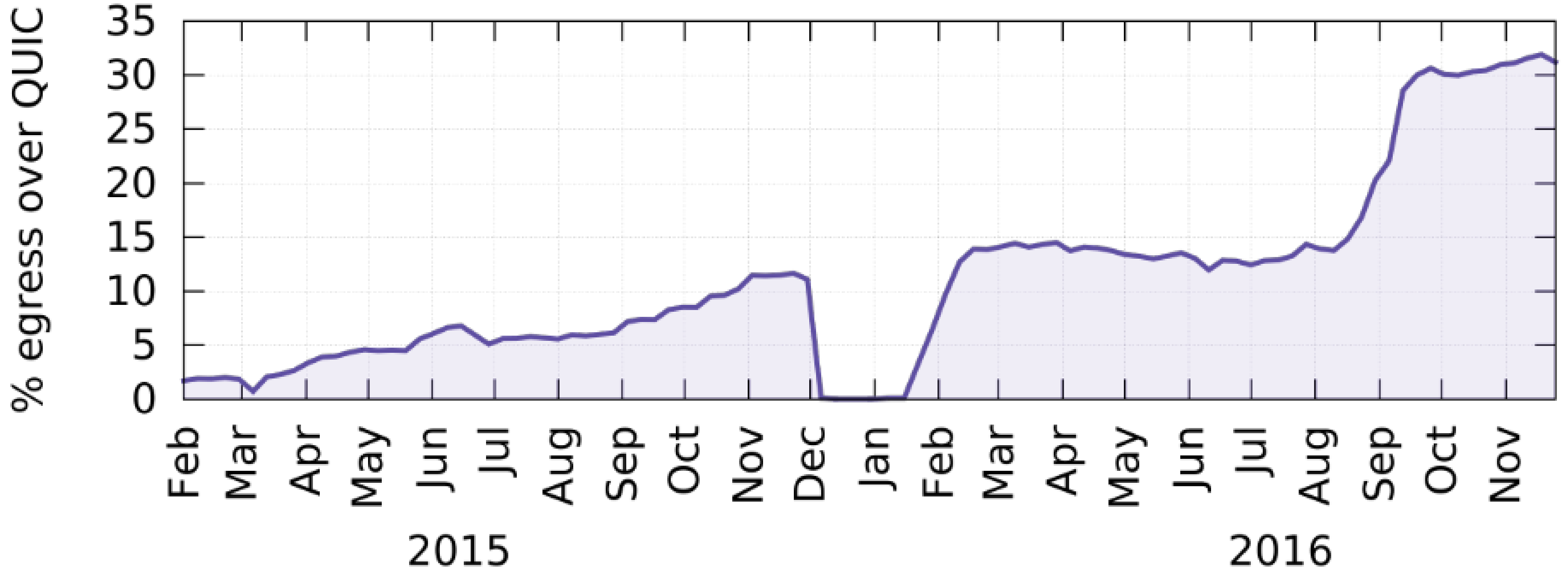


Figure 2: Timeline showing the percentage of Google traffic served over QUIC. Significant increases and decreases are described in Section 5.1.

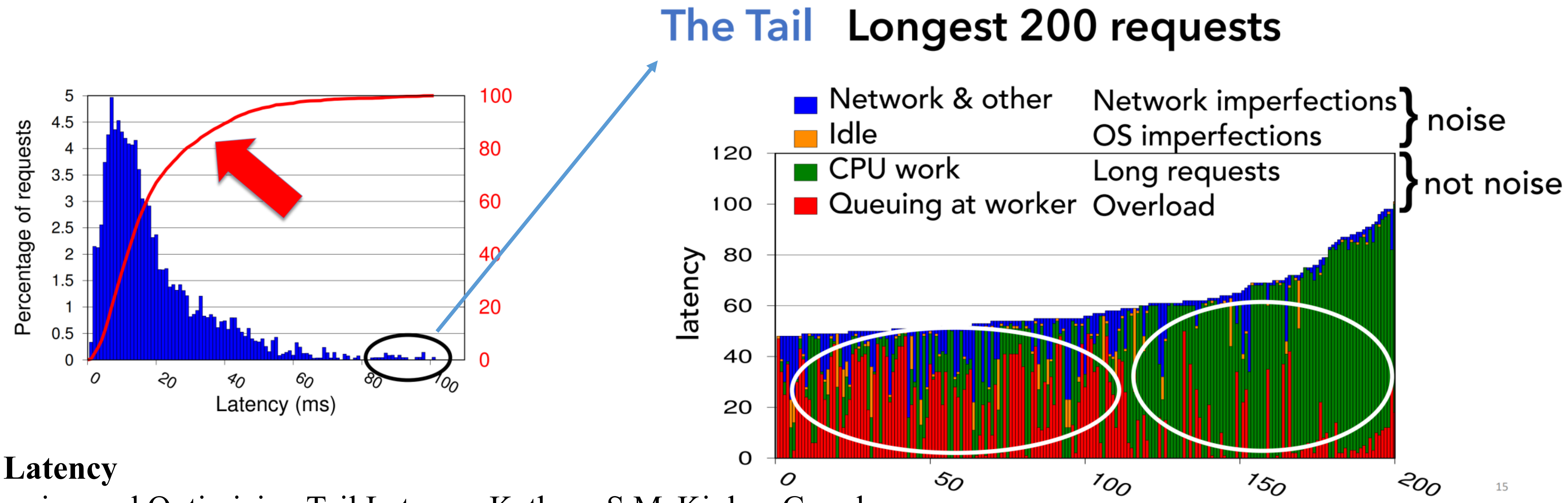
WHY QUIC

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].

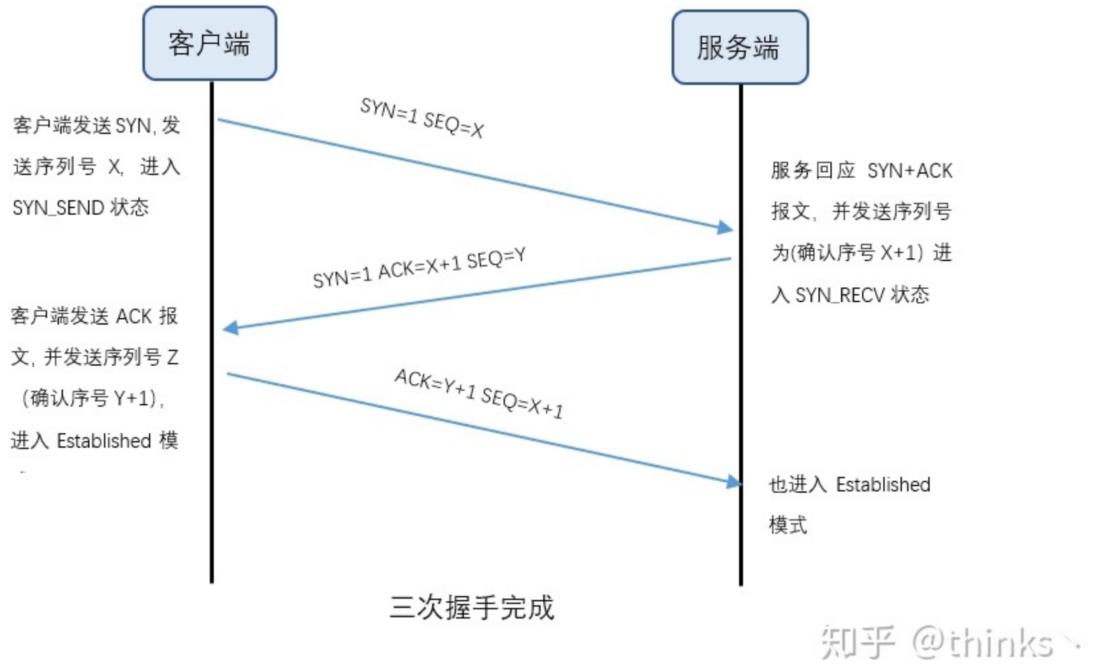
OS upgrades at servers tend to be faster by an order of magnitude but can **still take many months** because of appropriately rigorous stability and performance testing of the entire OS.

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.

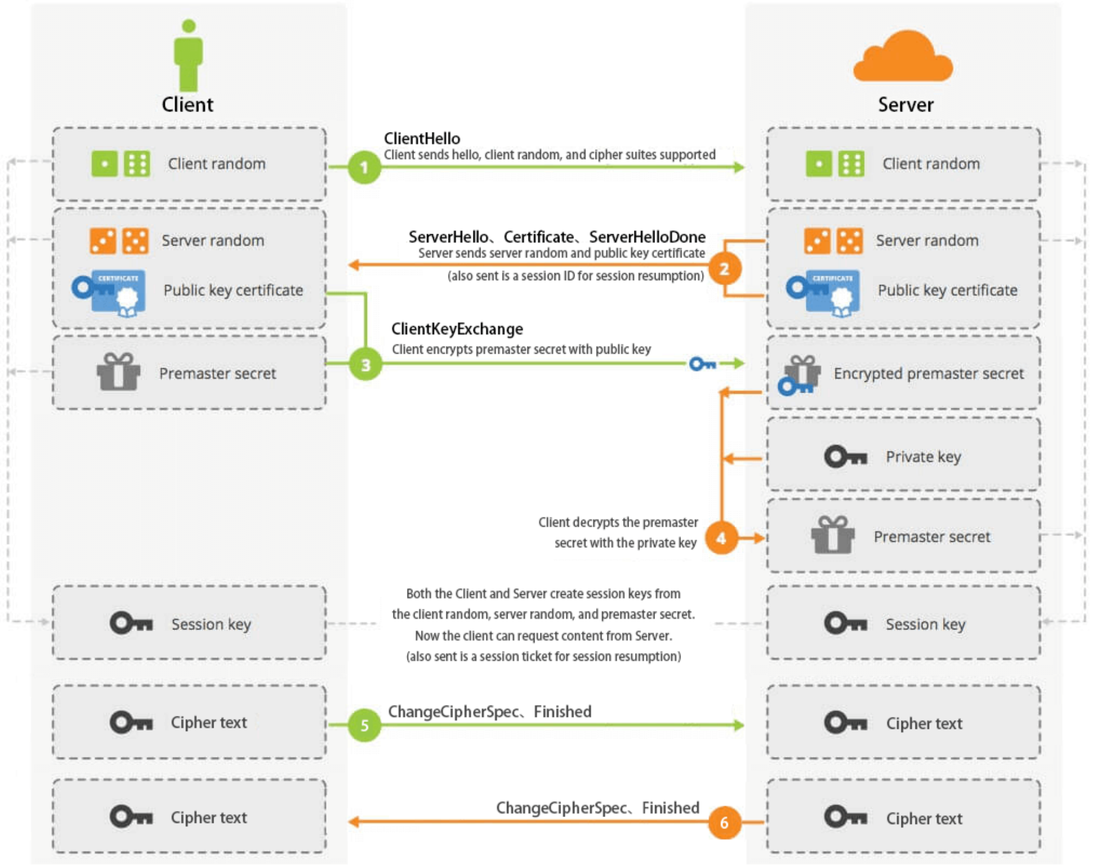
TCP 三次握手示意图


客户端
服务端
客户端发送 SYN, 发送序列号 X, 进入 SYN_SEND 状态
SYN=1 SEQ=X
服务回应 SYN+ACK 报文，并发送序列号为(确认序号 X+1) 进入 SYN_RECV 状态
SYN=1 ACK=X+1 SEQ=Y
客户端发送 ACK 报文，并发送序列号 Z（确认序号 Y+1），进入 Established 模
ACK=Y+1 SEQ=X+1
也进入 Established 模式
三次握手完成
知乎 @thinks丶

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

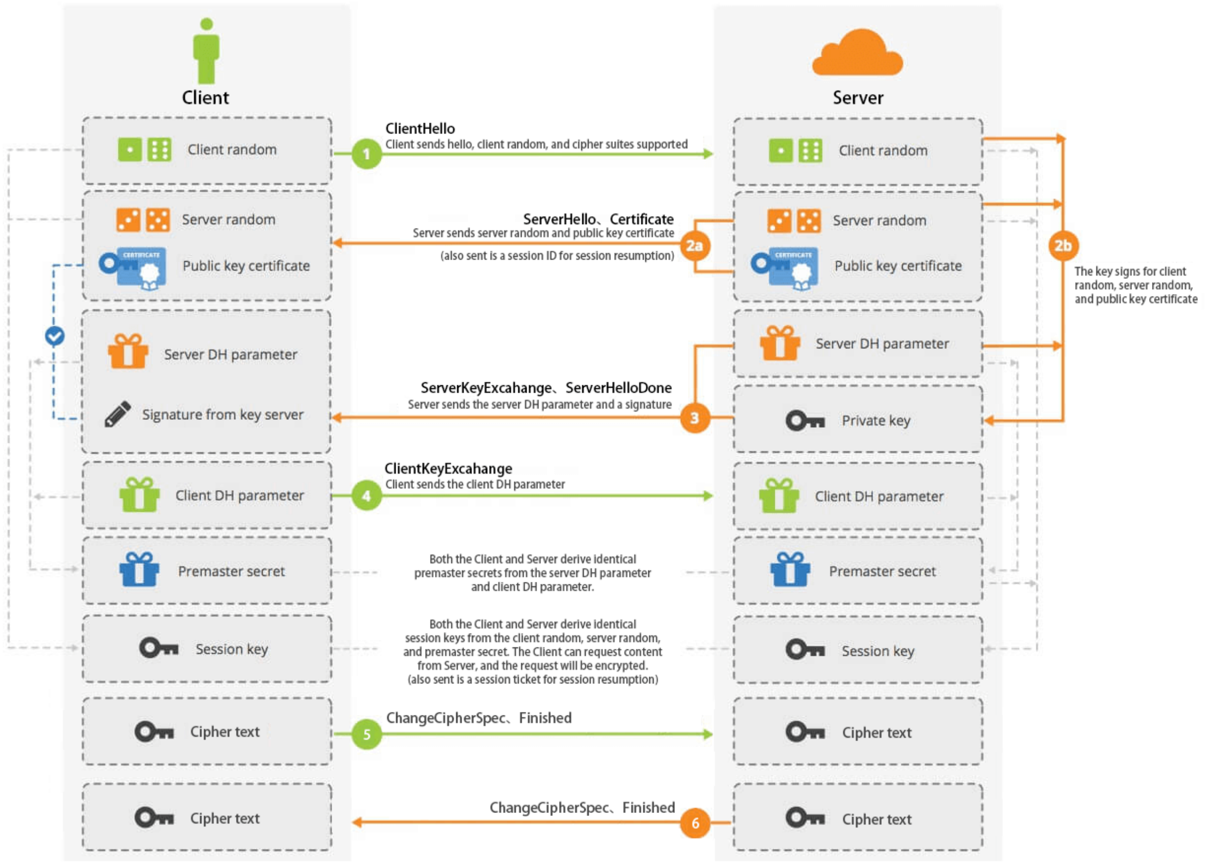
Handshake Delay

TLS Handshake RSA



Client
Server
Client random
Client random
1 ClientHello
Client sends hello, client random, and cipher suites supported
Server random
Public key certificate
Server random
Public key certificate
2 ServerHello、Certificate、ServerHelloDone
Server sends server random and public key certificate
(also sent is a session ID for session resumption)
3 ClientKeyExchange
Client encrypts premaster secret with public key
Premaster secret
Encrypted premaster secret
Private key
4 Client decrypts the premaster secret with the private key
Premaster secret
Session key
Session key
Both the Client and Server create session keys from the client random, server random, and premaster secret. Now the client can request content from Server. (also sent is a session ticket for session resumption)
5 ChangeCipherSpec、Finished
Cipher text
Cipher text
6 ChangeCipherSpec、Finished
Cipher text
Cipher text

TLS Handshake Diffie-Hellman



Client
Server
Client random
Client random
1 ClientHello
Client sends hello, client random, and cipher suites supported
Server random
Public key certificate
Server random
Public key certificate
2a ServerHello、Certificate
Server sends server random and public key certificate
(also sent is a session ID for session resumption)
2b The key signs for client random, server random, and public key certificate
Server DH parameter
Signature from key server
Server DH parameter
Private key
3 ServerKeyExcahange、ServerHelloDone
Server sends the server DH parameter and a signature
4 ClientKeyExcahange
Client sends the client DH parameter
Client DH parameter
Client DH parameter
Premaster secret
Premaster secret
Both the Client and Server derive identical premaster secrets from the server DH parameter and client DH parameter.
Session key
Session key
Both the Client and Server derive identical session keys from the client random, server random, and premaster secret. The Client can request content from Server, and the request will be encrypted. (also sent is a session ticket for session resumption)
5 ChangeCipherSpec、Finished
Cipher text
Cipher text
6 ChangeCipherSpec、Finished
Cipher text
Cipher text

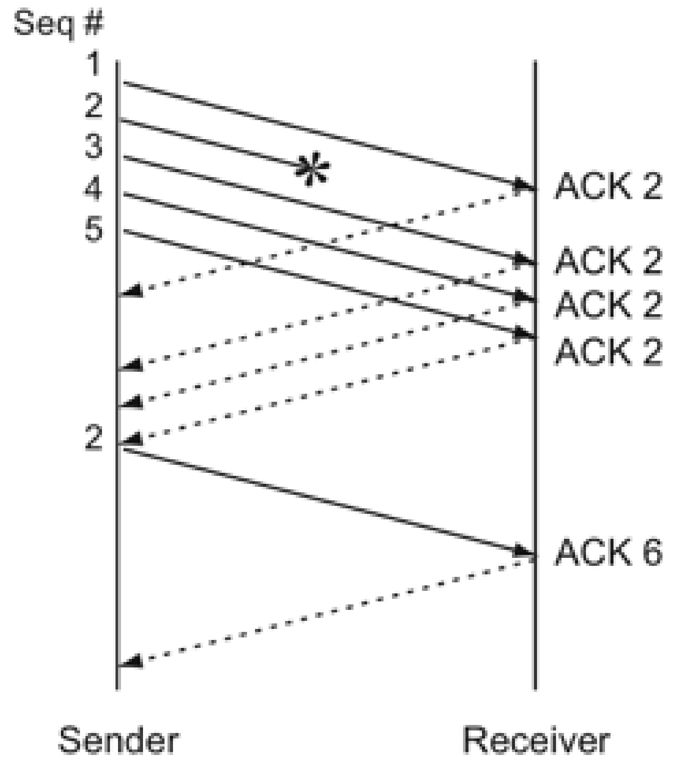
HTTPS 温故知新（三）—— 直观感受 TLS 握手流程(上)

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.

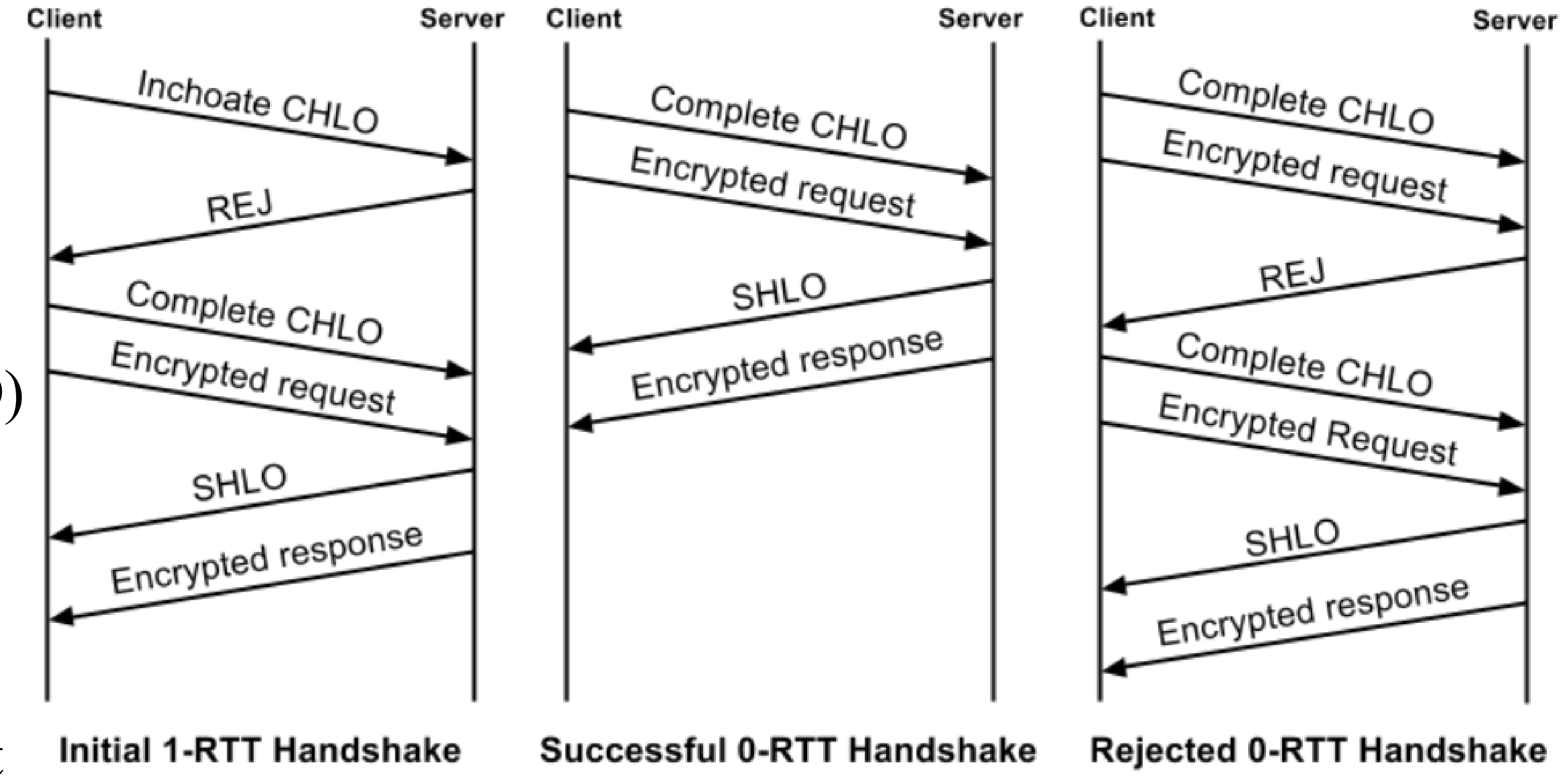


Figure 4: Timeline of QUIC's initial 1-RTT handshake, a subsequent successful 0-RTT handshake, and a failed 0-RTT handshake.

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.

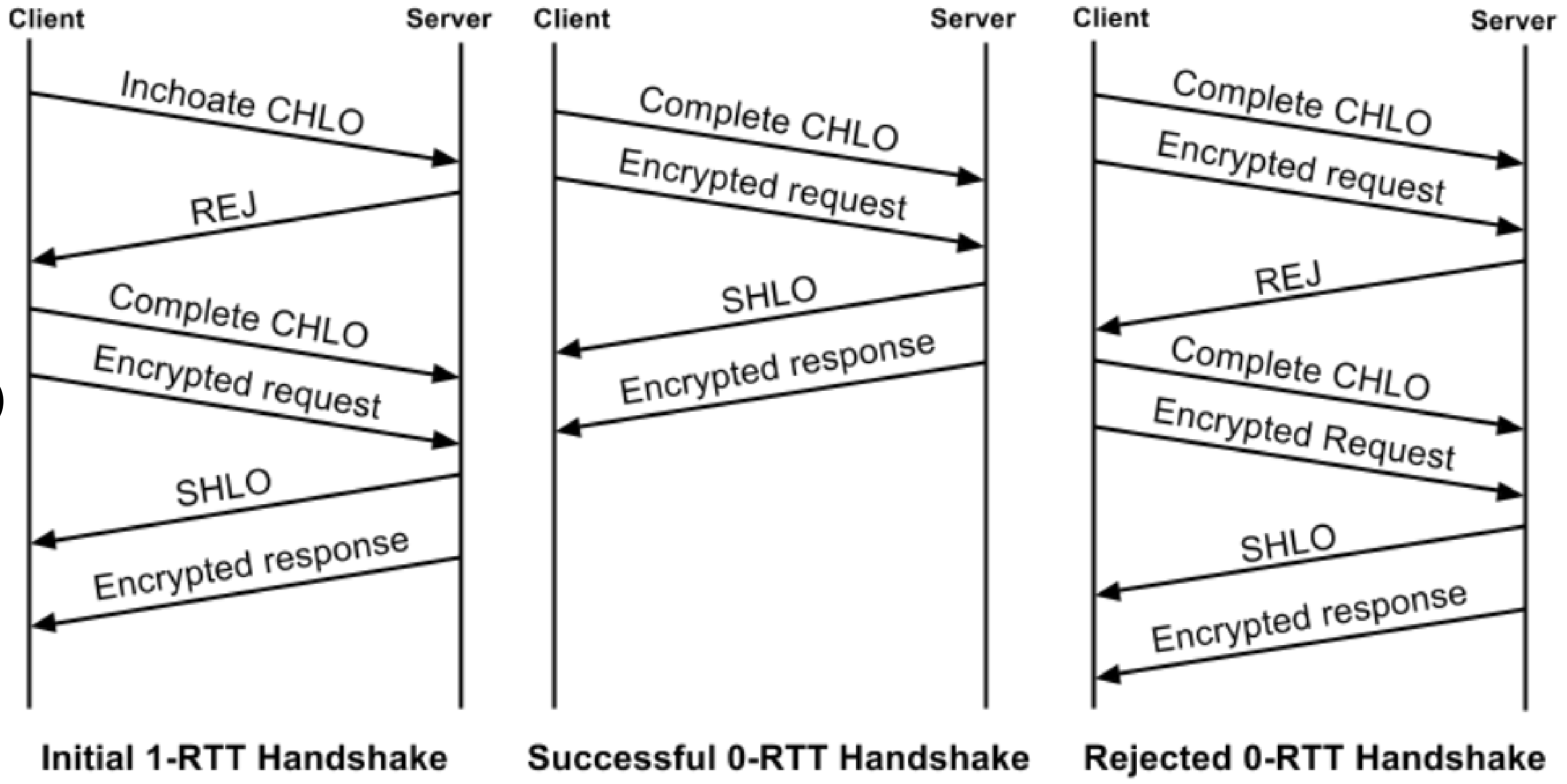


Figure 4: Timeline of QUIC's initial 1-RTT handshake, a subsequent successful 0-RTT handshake, and a failed 0-RTT handshake.

Connection Establishment

Version Negotiation

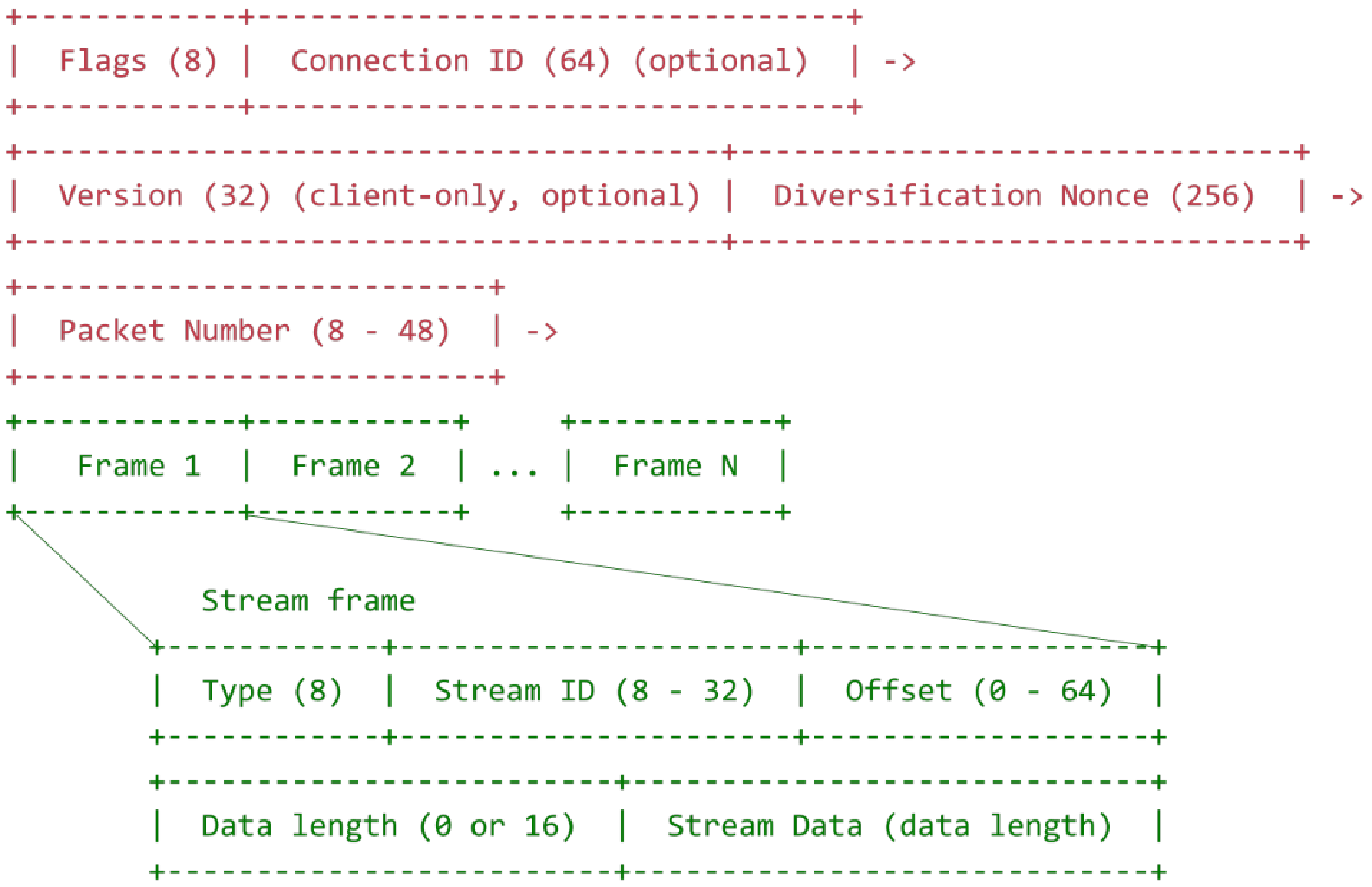
A QUIC client proposes a version to use for the connection **in the first packet of the connection** and encodes the rest of the handshake using the proposed version. If the server does not speak the client-chosen version, it forces version negotiation by **sending back a Version Negotiation packet** 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.**



```
+-----------+---------------------------------+
|  Flags (8) |  Connection ID (64) (optional)  | ->
+-----------+---------------------------------+
+---------------------------------+-----------------------------+
|  Version (32) (client-only, optional) |  Diversification Nonce (256)  | ->
+---------------------------------+-----------------------------+
+-----------------------+
|  Packet Number (8 - 48)  | ->
+-----------------------+
+-----------+----------+     +----------+
|   Frame 1  |  Frame 2  | ... |  Frame N  |
+-----------+----------+     +----------+

         Stream frame
       +-----------+-------------------+-----------------+
       |  Type (8)  |  Stream ID (8 - 32)  |  Offset (0 - 64)  |
       +-----------+-------------------+-----------------+
       +-----------------------+-----------------------+
       |  Data length (0 or 16)  |  Stream Data (data length)  |
       +-----------------------+-----------------------+
```

Padding Oracle Attack

Stream Multiplexing

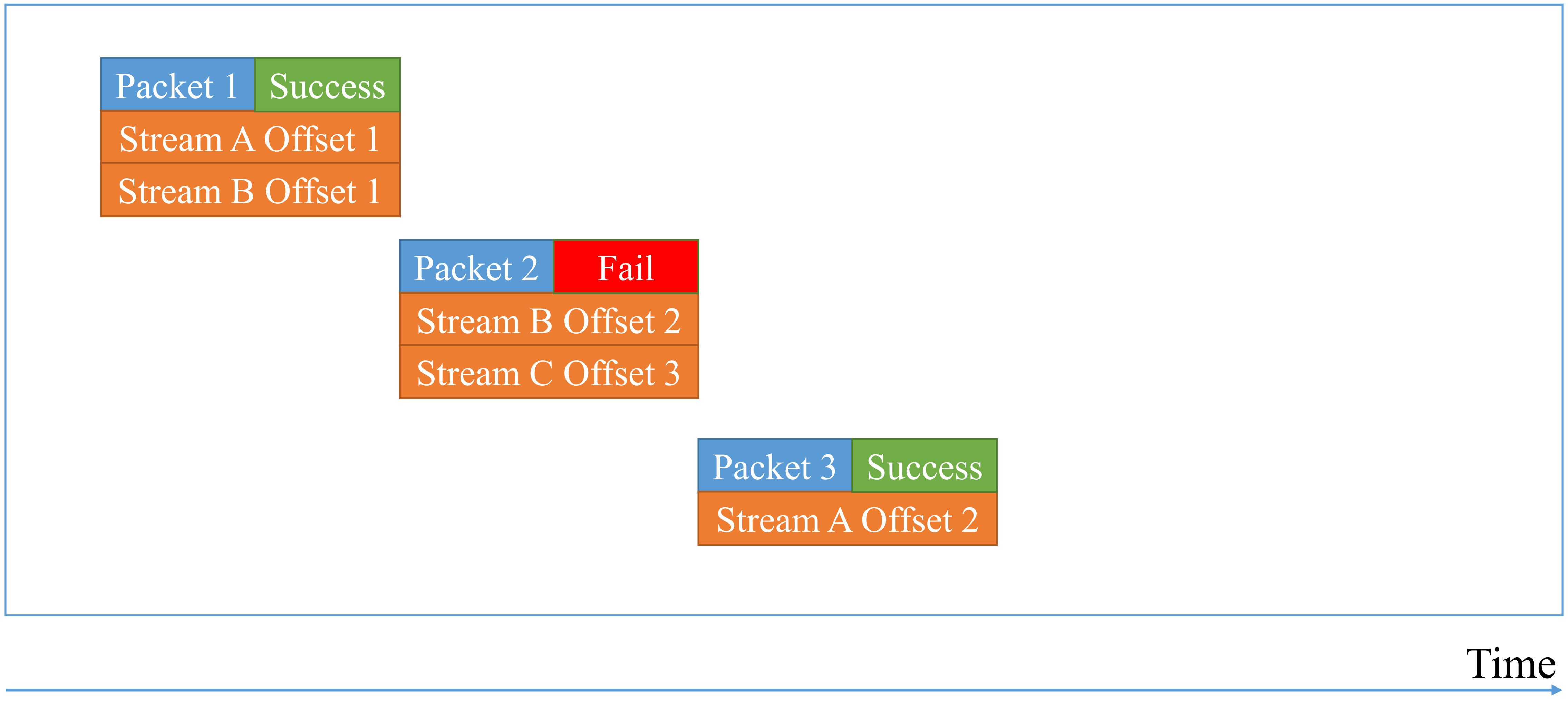
Packet Types and Formats

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.

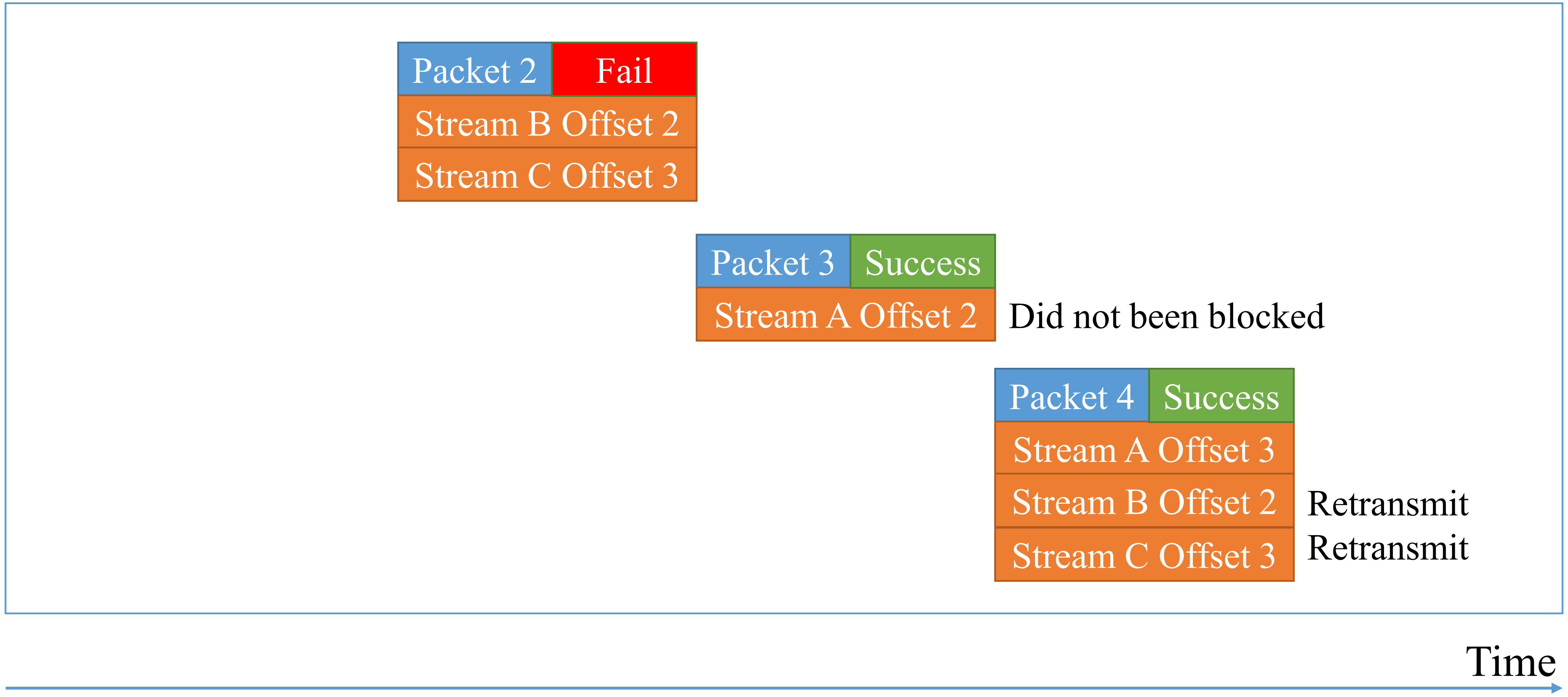
Frame Types and Formats

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.**

Stream Multiplexing



Stream Multiplexing



Authentication and Encryption

Cryptography - Diffie-Hellman

Suppose

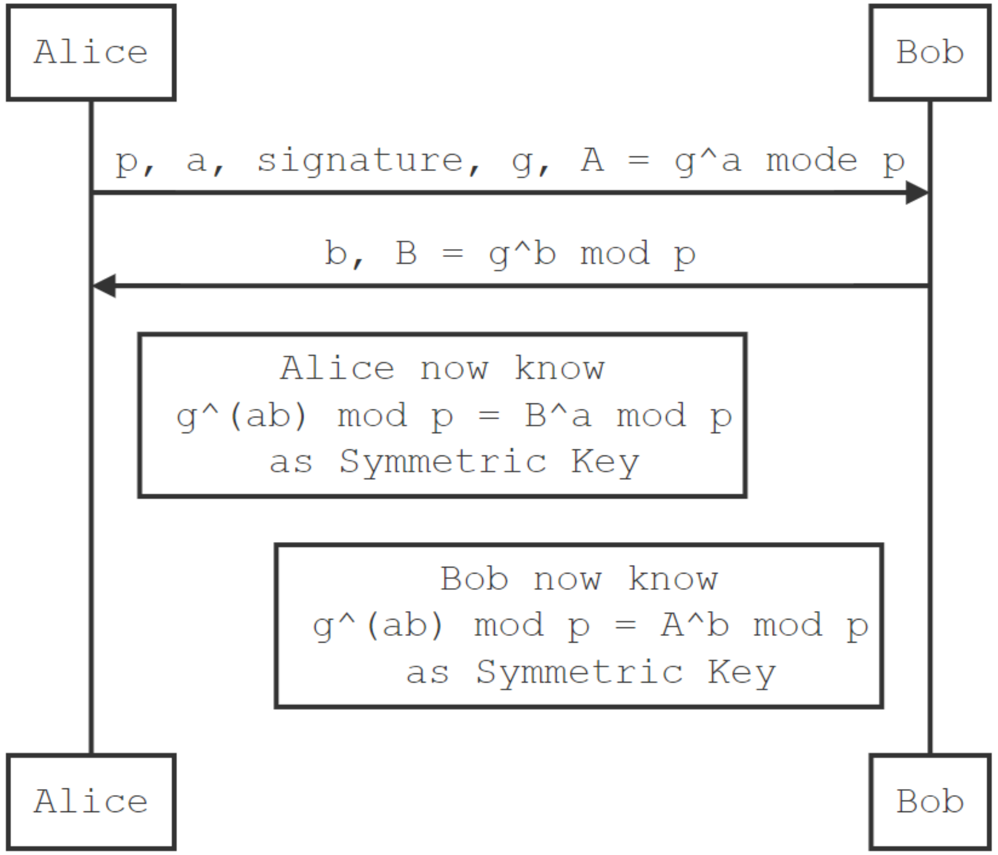
Client (Bob)

- *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

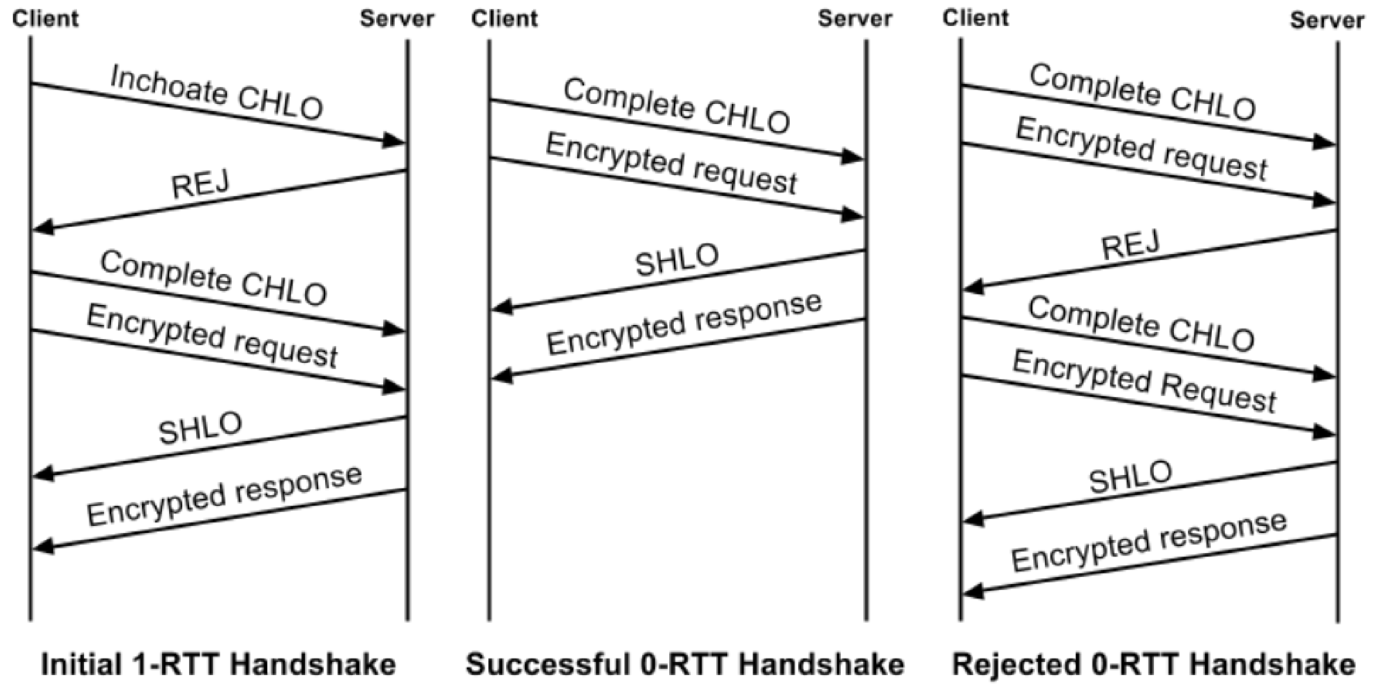
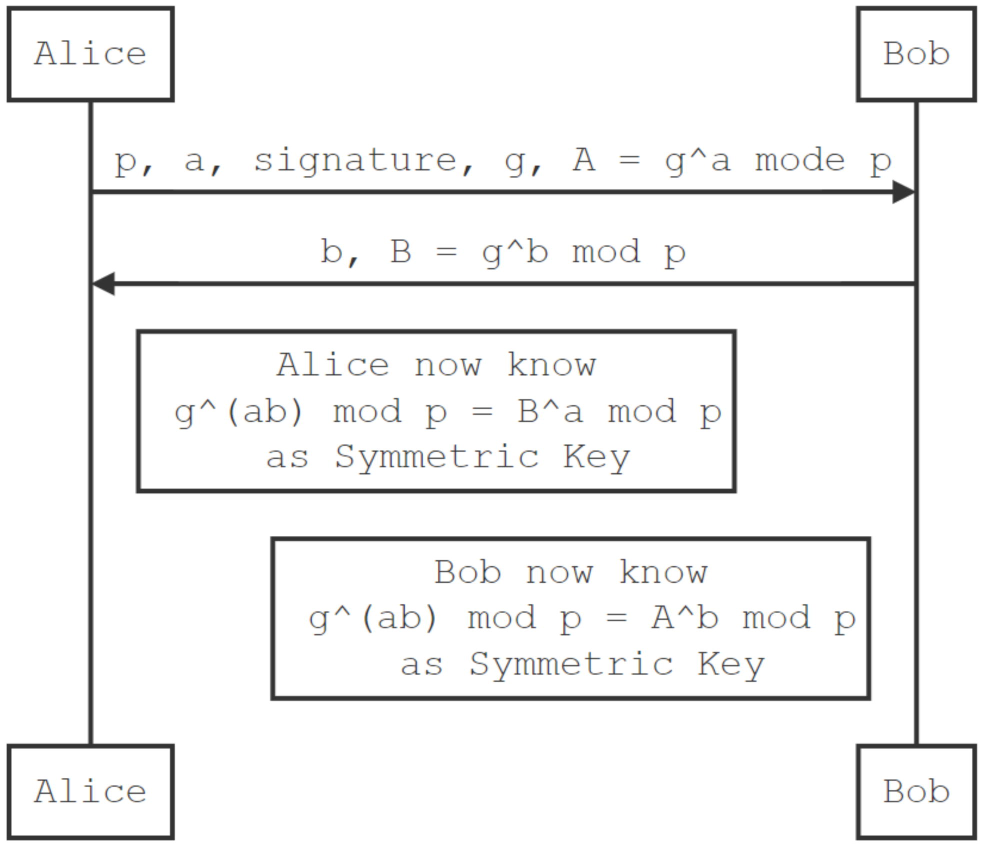


Figure 4: Timeline of QUIC's initial 1-RTT handshake, a subsequent successful 0-RTT handshake, and a failed 0-RTT handshake.



Loss Recovery

Proof of Diffie-Hellman

Flow Control

Proof of Diffie-Hellman

Congestion Control

Proof of Diffie-Hellman

NAT Rebinding and Connection Migration

Proof of Diffie-Hellman

QUIC Discovery for HTTPS

Proof of Diffie-Hellman

Open-Source Implementation

Proof of Diffie-Hellman

THANKS