A Quick Look at QUIC*

John Dellaverson, Tianxiang Li, Yanrong Wang, Jana Iyengar, Alexander Afanasyev, Lixia Zhang

ABSTRACT

QUIC represents the latest transport protocol development with the potential to replace TCP over time. Instead of describing the QUIC operations mechanically by enumerating, step-by-step, how it works, this paper aims to *explain* QUIC from the core ideas that its design is based on. We first describe TCP, the first deployed transport protocol and the most widely used so far, to explain the basic functions performed by a transport protocol and the issues observed from the TCP operations; we also discuss the insights learned from the design of a few other transport protocols. Then, we describe the design of QUIC in detail, with a focus on the fundamental underpinnings of its most important concepts, including its combination of connection and security setup, use of IP address independent connection identifiers and persistent connection state, separate window mechanisms for congestion control and data delivery reliability, and its tight coupling with applications, demonstrating how these new design features effectively address the identified issues in the existing transport protocols.

** This draft is still work-in-progress.*

1 INTRODUCTION

QUIC is the latest transport protocol with a big uptake. Some people predict that QUIC may eventually replace TCP over time [13]. Unfortunately, the QUIC specifications are not only long, but also largely focus on describing the protocol operations, with the explanation of *why* either embedded deeply in the protocol specifics or otherwise missing. For many people who want to learn about QUIC quickly, directly diving into the QUIC specifications does not seem an effective means to reach the goal. This writeup aims to offer a comprehensive and insightful look into QUIC.

QUIC looks very different from TCP that people are familiar with, but exactly what are the differences? And more importantly, why are these differences? Where did the new ideas come from? Although the QUIC design and development are still in progress, therefore various specific aspects may change in the future. Nevertheless, we expect that the core elements focused on by this paper will remain with the protocol.

We begin our explanation by examining TCP first, its basic functions, the issues that have been identified, and the lessons learned from years of transport protocol designs (§2). Then, we move into examining the QUIC protocol itself, separated into QUIC connections (§3), QUIC packets (§4), QUIC recovery (§5), QUIC security (§6), and application over QUIC (§7). Finally, we articulate the remaining challenges facing both QUIC and future transport protocols in general (§8).

2 TRANSPORT PROTOCOL 101

In this section, we use TCP as an example to identify the set of basic functions a (unicast) transport protocol must support. People familiar with TCP may skip §2.1 and §2.2 to go directly to §2.3, which describes a few other transport protocols that the QUIC design draws lessons from.

2.1 TCP

As a transport protocol, TCP performs the following three basic functions.

(1) demultiplexing of data by the use of port numbers;
(2) reliable byte stream delivery with window-based flow control[1]; and
(3) congestion control to limit the number of *packets* inside the network.

First, the TCP/IP protocol stack bridges the gap between semantically meaningful application/service names and the lower layer protocols by making use of port numbers. Each standard application or service is assigned a specific transport protocol port number [5, 12]. All the existing transport protocols use the combination of source and destination port numbers to deliver incoming packets to the right application process.

Second, to provide reliable data delivery service between two endpoints, every data piece, in TCP's case every data byte, must be assigned a unique identifier, which allows the two ends to figure out whether all the data pieces have been delivered. All transport protocols achieve this goal by first defining a unique connection identifier, and within the connection assigning each data unit a unique identifier, which is a monotonically increasing sequence number in general. With monotonically increasing sequence numbers, the receiver can inform the sender of all the data it has received up to Seq# n by a simple cumulative acknowledgment $ACK(n)$.

[1]In order to provide a generic reliable delivery service for different applications, TCP chose to use *byte* as its basic data unit, ensuring an in-order delivery of byte stream.

TCP uses the combination of the two endpoints' IP addresses and port numbers to create a globally unique connection identifier, and a reliable connection setup process to let both ends agree on the initial sequence number to be used in starting data communication. TCP also uses a similar reliable tear down process to let each end inform the other of the final sequence number that identifies the last byte of the data it sends to the other. Both setup and tear-down processes follow a standard 3-way handshake.

For connection setup, to minimize the chance of the sequence number used by this new connection colliding with that used in previous connections[2], TCP lets each of the two ends pick a random number as its initial sequence number. In this case, the initiating end (client) can start sending data with the third handshake message, thus there is one round trip delay to inform each other of the initial sequence number.

Third, the goal of congestion control (CC) is to limit the number of packets *inside the network*. TCP was not designed to perform CC per its specification [1], TCP implementation was revised in mid 80's to add congestion control on top of its window-based flow control mechanism. TCP-CC lets the sender maintain a *congestion window*, and send data within the constraint of $MIN[flow\ window, congestion\ window]$.

2.2 TCP's Problems

In the following we enumerate the major issues that have been identified from the use of TCP over the years.

1. **The coupling between CC and reliable delivery** CC's goal is to control the number of packets *inside the network*, and function was latched onto the same window mechanism which was originally designed for receiver to perform flow-control on the sender, and to assure reliable delivery. Therefore any packet loss will constrain the window advancement until the loss is recovered; during the time period of loss detection and data retransmission, the majority, if not all, of the packets within the window may have been delivered to the receiver node, i.e. have exited the network. As shown in Figure 1, where the receiver sets the flow control window size to be equivalent to 8 packets; we can assume CC window is also 8 packets. As packet 1's arrival advances the window to enable the transmission of packets 2–9; the loss of packet 2 stops the window from moving forward, even when no packet inside the network. Therefore the number of unacknowledged packets does not reflect the number of packets inside the network. A few patches of "window inflation, deflation" haven been developed to mitigate this problem, with added complexity and limited effectiveness [3].

[2]This is to avoid mistaking the data from previous connection C_{prev} for the new connection, in case C_{prev} happens to have the same port numbers and IP addresses.

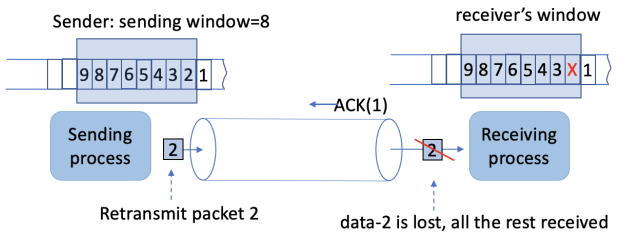

Figure 1: All packets are out of the network; the loss of packet-2 prevents flow window from moving forward.

2. **Head of line blocking** (HLB): because TCP assures sequential delivery of data byte streams, the loss of one single packet will block the delivery of all subsequent packets until the lost packet has been recovered. Fig. 2 shows an example where packet 2 has been delivered to the receiver but packet 1 is lost during transit. The data of packet 2 cannot be delivered to the application until packet 1 has been retransmitted and received, and passed to the application first, to assure sequential delivery of all data to the application process. Figure 1 and Figure 2 look similar, with the former reflecting an issue at the sender (which is blocked from sending more packets), and the latter an issue at the receiver (already received data gets blocked inside the kernel).

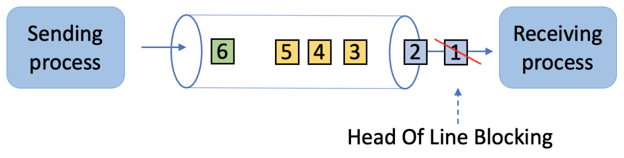

Figure 2: TCP Head of Line Blocking.

3. **Delay due to connection setup:** Every time an endpoint wants to communicate with another via TCP, a 3-way handshake is required to set up a TCP connection before application data can be sent. And if one wants to secure this connection by TLS, another round trip is needed for the two ends to exchange security credentials.
4. **Limitations due to fixed protocol header** : TCP header is a collection of *fixed-size* fields. It does have an option field, which is limited to 40 bytes max. In addition to carry application data between the two communicating ends, the TCP protocol design also packs all the control functions into the same 20-byte TCP header:

connection setup, tear down, and reset; and data acknowledgment. When additional functions are identified, e.g., Selective ACK, they are added into the option field which has its own length limit of 40-byte maximum.

Three specific fields in the TCP header have been directly affected by the continued increase of network speed over time. The sequence number and ACK fields each are 4-byte long, and the flow control window size is 2-byte only. These small, fixed size fields limit TCP's performance at high network speed: the first two wrap around too quickly, making them no longer unique; and a small size of flow control window directly limits a TCP connection's throughput which is upper bound by ($window\ size \times RTT$). Again the TCP header option has been used to extend their lengths.

5. **Unique connection identifier and IP address** The IP address of either end host in a TCP connection may change during the connection's lifetime, due to a variety of reasons: host multihoming, mobility, or running behind an NAT. Because TCP uses the combination of two endpoints' IP addresses and port numbers as the connection identifier, any change in either IP address would break the existing connection, resulting in all the data exchanged up to that point being thrown away. To recover from the failed TCP connection, a new 3-way handshake is required to set up a new connection.

2.3 Other Transport Protocols

Although TCP has been the dominant transport protocol in use by far, several other transport protocols have also been developed over the years, each filling some needs missing from TCP.

2.3.1 ***T/TCP****.* Transaction TCP[4] was developed to support distributed applications with frequent realtime transactions, without paying the cost of delay and overhead from TCP's connection setup process. T/TCP keeps the connection state after the initial establishment, namely the connection identifier with associated sequence number of both ends. Keeping a persistent connection state adds additional system memory cost, but avoids the 3-way handshake delay for subsequent short transactions, which may be separated by long idle time periods in between.

2.3.2 ***SCTP****.* The design of Stream Control Transmission Protocol (SCTP) [18] preceded QUIC by 10+ years. Its design addresses three of the identified TCP issues.

For TCP **issue #2** Head-of-Line-Blocking, SCTP allows each SCTP connection to contain multiple data substreams, removing the head-of-line blocking problem *between* the data belong to different substreams. A lost packet of substream-1 will not block SCTP from passing to the application the received data for substream-2, confining the HOL problem to be with each substream, which is still delivered in-order. However, SCTP does not fully address the HLB problem. Each SCTP connection uses a single TSN (Transmission Sequence Number) for both reliable delivery and congestion control, in the same was as TCP does. Therefore, when the TSN window is blocked by lost packets, the sender cannot send any new data for any substream, suffering a similar problem as TCP.

For **TCP issue #4**, SCTP defined multiple *typed chunks*, each chunk has its own defined header formate; and multiple chunks can be packed into a single SCTP packet within the MTU limit. Each connection management command, e.g. setup, tear-down, reset, etc., is defined as a separate control chunk, so is the Selective ACK. Application data are carried in *data chunks* as ADUs, each identified by [stream ID, stream seq#][3]. This design gives SCTP flexibility of defining new connection management commands by simply adding a new chunk type.

SCTP partially addressed **TCP issue #5** on IP address changes. Each end of a SCTP connection (called an association) can have multiple IP addresses, and the set may change during the connection's lifetime. Nevertheless, SCTP connection identifiers are still bound to IP addresses.

2.3.3 ***RTP****.* As its name suggested, Real-time Transport Protocol (RTP) [9] is designed to support *real time* applications, such as video conferencing. Real-time multimedia streaming applications require timely delivery of data, and to achieve this primary goal it can tolerate packet losses to certain degree. The protocol also provides facilities for multicast packet delivery and jitter compensation. Due to its radically different purposes and requirements from previous transport protocols which focus on *reliable* data delivery, RTP pioneered a few new concepts in transport protocol designs.

First, RTP is the first widely used transport protocol that runs over UDP, resulting in RTP implementation being *outside the kernel.* UDP [16] can be viewed as a NO-OP transport protocol other than using port numbers to demultiplex incoming packets to intended application processes, and an optional checksum. However, UDP does offer a likely unnoticed but a rather important advantage: it allows applications to control what data to be packaged into each IP packet, thus enabling the realization of the *application data unit (ADU)* [7]. This user space implementation allows RTP to be tightly integrated with the application implementations to utilize ADU.

[3]Note that each stream seq# identifies a data chunk given by the application, whose size can be larger than the network MTU. Therefore SCTP needs to handle data chunk fragmentation and reassembly, in contrast to TCP's byte stream data model, where each seq# identifies a data byte which allows TCP to chop an application data segment at any byte boundary.

Second, RTP is the first widely used transport protocol that uses IP multicast delivery. A multicast RTP session is identified by the IP multicast address plus a pair of UDP port (one for RTP, one for RTCP). Furthermore, RTP data packets carry timestemps and sequence numbers, the former for media replay and the latter for detecting losses.

2.4 Securing Transport Protocols: TLS and DTLS

Internet applications need crypto protection, which has been patched onto the existing transport protocol.

The primary advantage of the Transport Layer Security (TLS) protocol is that it provides a transparent connection-oriented channel. Thus, it is easy to secure an application protocol by inserting TLS between the application layer and the transport layer, typically TCP. TLS encrypts all the data exchanges of a TCP connection, providing authenticity and confidentially protection between two communicating end points [17]. TLS runs in the user space and relies on TCP to provide reliable delivery for its own structured data units used for data encryption. A handshake process is required for TLS to establish cryptographic parameters needed for the encryption before the application data can be sent over the TCP connection.

Datagram Transport Layer Security (DTLS) protocol [8] is designed to secure applications that use UDP as the transport protocol. The basic design philosophy of DTLS is to construct "TLS over datagram transport". TLS requires reliable packet delivery and cannot be used directly in datagram environments where packets may be lost or reordered. DTLS makes only the minimal changes to TLS required to fix this problem. DTLS uses a simple retransmission timer to handle packet loss, a sequence number to handle network packet re-ordering, and performs its own message fragmentation and reassembly when needed. In essence, DTLS performs all the tasks of TLS with its own TCP-equivalent reliability support. As a result, using DTLS to secure application datagrams require multiple round trips.

2.5 Summary: Transport Protocol Functions over IP

Based on the observation of the transport protocol designs and their usages over the last few decades, below we make a summary of the basic functions that a unicast transport protocol running over IP needs to provide and the related design questions. Note that all the existing transport protocols use port numbers for data demultiplexing inside a host in a straight forward way, thus we omit it in the following enumeration.

1. defining connection identifier and data identifier

2. transport connection management

- **Globally unique connection ID** that binds the two ends of the connection, wishes to be IP address independent but must map to IP addresses reliably.
- connection state setup and tear down
- **Control information exchange** between the two ends; wish to have flexibility in defining new control messages.
- support of host IP address changes (can be due to host multihoming, physical mobility, or other causes)

3. reliable data delivery

- **Unique data identifier** that must be reliably exchanged with the other end.
- **Window flow control for reliable delivery**; wish to avoid the head-of-line blocking problem.

4. Congestion control: [4] needs to control the number of packets inside the network.

5. Security To be more specific, people currently view encrypted connections as *network security*, although TLL encrypted channels provide only data confidentiality; remote party authenticity and trust are managed through third-party certificate authorities (CAs).

Connection establishment and security have been separate steps, e.g. setup a TCP connection first to enable reliable data delivery, then set up TLS association on top of it. DTLS implicitly combines the two.

In addition, transport protocols have traditionally been implemented inside the kernel, one reason is for performance efficiency. Every coin has two sides, here the other side of the story is the lack of control on the data packaging and difficulty in making protocol changes. RTP's way of running over UDP makes it outside the kernel and supports ADUs. As we see next, QUIC adopted the same approach.

3 QUIC CONNECTION

QUIC is a transport protocol originally designed by Google to improve transport performance for encrypted traffic and to enable rapid deployment and continued evolution of transport mechanisms. [6] QUIC uses UDP as an underlying protocol and its implementation runs in the user space, making it easy for future updates without having to wait for a kernel upgrade. Learning the lessons from previous protocol designs, QUIC addressed TCP's problems identified in §2.2. A QUIC connection is a shared state between a client and a server, which always starts with a handshake process and during which the two endpoints establish the parameters for the connection [15].

[4] Ideally CC should be a network layer function, it landed on TCP because IP is open-loop and thus has no effective means to control packet transmission.

3.1 Connection ID

QUIC uses the combination of two numbers, one selected by each end, to form a pair of connection IDs. The connection ID (CID) acts as a unique identifier for the connection, which is used to ensure that changes in addressing at lower protocol layers will not cause packets to be delivered to a wrong recipient. By using the connection ID, QUIC supports the demultiplexing ability similar to the functionality provided by TCP.

Related RFC: QUIC-TRANSPORT[15] §5 Connections.

3.2 QUIC Handshake

QUIC combines transport and cryptographic handshakes together, acquiring the information necessary for both in 1-RTT. More specifically, this entails doing an authenticated TLS 1.3 key exchange along with an authenticated transport parameters exchange at the same time. This minimizes the latency necessary to set up a secured connection. Whereas conventional TCP keeps security and transport parameter exchanges separate, requiring at least 2 RTTs to set up a secure connection.

QUIC uses the Initial packet to negotiate the connection IDs for a new connection. Each endpoint will populate the Source Connection ID field with its chosen value in its Initial packet and that ID will be used by the other endpoint to set the Destination Connection ID when sending future packets. Upon receiving an Initial packet, a server can optionally choose to verify a client's address by sending a Retry packet containing a random token, which should be repeated by the client in a new Initial packet to continue the handshake process. TLS 1.3 handshakes messages are also embedded in these Initial packets, which could establish a shared secret to protect the confidentiality and authenticity of future packets in 1-RTT. The chosen connection IDs will be included in the QUIC transport parameters, which will be authenticated during the TLS handshake process. With the negotiation of connection IDs, QUIC supports the reliable setup of a new connection similar to the functionality provided by TCP.

QUIC allows a client to send 0-RTT encrypted application data in its first packet to the server by reusing the negotiated parameters from a previous connection and a TLS 1.3 pre-shared key (PSK) identity issued by the server, though these 0-RTT data are not protected against replay attack. By supporting sending 0-RTT data, QUIC is also able to handle use cases where T/TCP is required. More detail about the cryptographic part of the handshake process is discussed in §6.1.

Related RFC: QUIC-TRANSPORT[15] §7 Cryptographic and Transport Handshake and §8 Address Validation, The Transport Layer Security (TLS) Protocol Version 1.3 [17] §2 Protocol Overview and §4 Handshake Protocol.

3.3 Connection Migration

Unlike TCP where the combination of the two endpoints' IP addresses and port numbers are used as the connection identifier, QUIC connection has the ability to survive changes in underlying protocol addresses with the usage of Connection ID. After a network change, a migrating endpoint can send a packet with previously established connection IDs using its new address to initialize the connection migration process. After the other endpoint received that packet, it will perform path validation to verify the peer's ownership of the new address by sending a special challenge frame containing some random data to the peer's new address and waiting for an echoed response with the same data, and the two endpoints can continue to exchange data after the verification of the new address. To prevent a passive observer from correlating the activity of an endpoint between different network paths, a QUIC endpoint can provide its peer with alternative connection IDs in advance and a migrating endpoint can use different connection IDs when sending data from different addresses.

QUIC also allows a server to accept connections on one IP address and ask the client to migrate to a new server address by using the transport parameters to convey its preferred address during the handshake process. After the handshake is confirmed, the client will perform a path validation on the server's preferred address, and once it succeeds, it will send all future packets to the new server address.

Related RFC: QUIC-TRANSPORT[15] §8 Address Validation and §9 Connection Migration.

4 QUIC PACKET

4.1 Packet Format

Unlike TCP where the packet header format is fixed, QUIC has two types of packet headers. QUIC packets for connection establishment need to contain several pieces of information, it uses the long header format. Once a connection is established, only certain header fields are necessary, the subsequent packets use the short header format for efficiency [13]. The short header format that is used after the handshake is completed is demonstrated in Fig. 4. In each packet, one or more frames can be embedded in it and each frame does not need to be of the same type as long as it is within the MTU limit.

Each packet in a QUIC connection is assigned a unique packet number. This number increases monotonically, indicating the transmission order of packets and is decoupled from loss recovery [5]. Therefore, it can be used to tell easily and accurately about how many packets may be inside

[5]QUIC uses different packet number space for each encryption level (initial packets, handshake, application data). Packet numbers are unique within each packet number space, and packets are acknowledged in their own

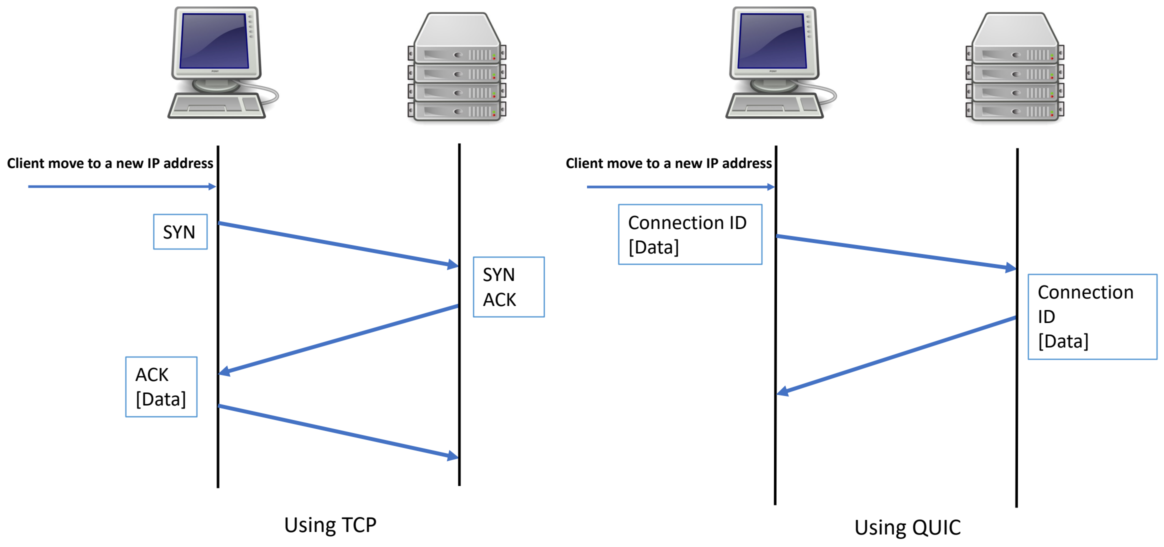

Figure 3: Comparison of TCP and QUIC after having an IP address change. When using TCP, the old connection will be discarded and a new three-way handshake taking 1-RTT will be required before application data can be exchanged. While in QUIC the old connection can be reused and the two peers can directly start sending data if the server has verified the client's address in the past (for example, when the client is moving back to an old address).

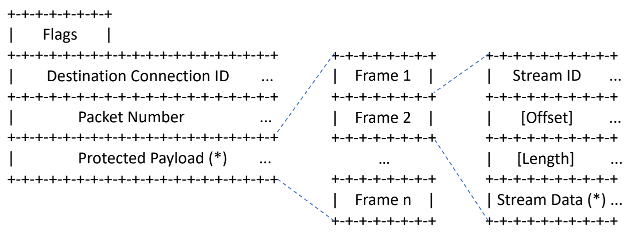

Figure 4: QUIC Short Header Packet Format. Frame 2 is a Stream frame containing application data.

the network, as compared to TCP congestion control which shares the same flow control window used for reliability.

QUIC receiver ACKs the largest packet number ever received, together with selective ACK (ACKing all received packets below it, coded in continuous packet number ranges) as shown in Figure 5. The use of purposely defined ACK frames can support up to 256 ACK blocks in one ACK frame, as compared to TCP's 3 SACK ranges due to TCP option field size limit. This allows QUIC to ACK received packets repeatedly in multiple ACK frames, leading to higher resiliency against packet reordering and losses. When a QUIC packet is ACKed, it indicates all the frames carried in that packet have been received.

packet number spaces. This enables cryptographic data separation between different packet spaces.

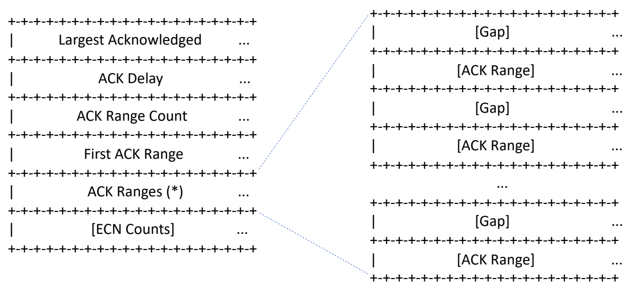

Figure 5: QUIC ACK Frame Format. The Largest Acknowledged field indicates the largest packet number the sender is acknowledging. An ACK Range indicates the number of continuously acknowledged packets before the largest acknowledged packet number. The gap indicates the number of continuously unacknowledged packets before each ACK Range

Related RFC: QUIC-TRANSPORT[15] §12 Packets and Frames, §13 Packetization and Reliability, and §17 Packet Formats.

4.2 Stream

QUIC has adopted several features directly from HTTP/2, and one of them is incorporating stream multiplexing into the transport layer. In the same way that in HTTP/2, multiple streams can exist on one TCP connection, each QUIC connection can have multiple simultaneous flows. This idea also borrows from the structured stream abstraction of SST [11]. Besides, QUIC has also adopted the idea of chopping data into frames and uses those as the basic unit of communication.

Each QUIC stream is identified by a unique stream ID, where its two least significant bits are used to identify which endpoint initiated the stream and whether the stream is bidirectional or unidirectional. Each stream resembles a TCP connection, providing ordered byte-stream delivery. The byte stream is cut to data frames, analogous to TCP segments. **Stream frame offset** is equivalent to TCP seq#, used for data frame delivery ordering and loss detection and retransmission for reliable data delivery. Each data frame is uniquely identified by [stream ID, frame offset]. QUIC uses the STREAM frames to transmit application data and multiple frames from different streams can be packaged into one QUIC packet for transmission.

QUIC endpoints can decide how to allocate bandwidth between different streams, how to prioritize transmission of different stream frames based on information from the application. This ensures effective loss recovery, congestion control, flow control operations, which can significantly impact application performance.

Head-of-Line Blocking: The HTTP/2 protocol used stream multiplexing to solve the HTTP HOL Blocking problem (A HTTP client can only open a limited number of concurrent TCP connections to a server, and when that limit is reached any subsequent requests need to wait for a previous request to finish). However, because HTTP/2 is multiplexing over a single TCP connection it will still suffer from the TCP HOL Blocking problem.

Since QUIC uses multiple independent streams, it avoids the Head-of-Line Blocking problem caused by waiting for recovering lost packets in TCP. When a packet is lost, only the streams with data frames contained in the packet will need to wait for the retransmission of the lost frames. It will not block other streams from moving forward. For instance, in Fig. 6 there are three QUIC streams denoted in red, green, and blue for a single connection. In case there is a packet loss for the red stream, it will not block the delivery of packets for the green stream and the blue stream.

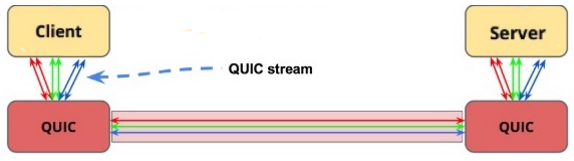

Figure 6: QUIC Solving HOL Blocking Problem.

Related RFC: QUIC-TRANSPORT[15] §2 Streams and §3 Stream States.

4.3 Unreliable Datagram Delivery

Some applications, particularly those that need to transmit real-time data, prefer to transmit data without reliable delivery. Currently one may support these applications by using UDP as the transport protocol, or the secure counterpart DTLS [RFC6347]. QUIC supports the unreliable but secured data delivery with the DATAGRAM frames, which will not be retransmitted upon loss detection[10]. With the support of the unreliable datagram, QUIC could improve the above approach with a reliable and authenticated handshake, followed by secure but unreliable delivery of application datagrams. QUIC packets containing only DATAGRAM frames are also ACK-eliciting, so the application can track whether a DATAGRAM frame is delivered or not.

Related RFC: QUIC-DATAGRAM[10] §3 Transport Parameter, §4 Datagram Frame Type, and §5 Behavior and Usage.

5 QUIC RECOVERY

5.1 Estimating the Round-Trip Time

QUIC ACK frames encode the delay between the receipt of a packet and the transmission of its ACK, which allows the receiver of the ACK to calculate the actual time used in transmitting a packet over the network. So when receiving an ACK frame, a QUIC endpoint can generate an RTT sample of the network path by calculating the time elapsed since the largest acknowledged packet was sent.

QUIC uses the following three values to generate a statistical description of a network path's RTT: the minimum RTT (`min_rtt`), an exponentially weighted moving average RTT (`smoothed_rtt`), and the mean deviation of the observed RTT samples (`rttvar`). With the usage of a monotonically increasing packet number, QUIC retransmission avoids the "retransmission ambiguity" problem in TCP, which is caused by the retransmitted packet carrying the same sequence number as the lost packet.

Related RFC: QUIC-RECOVERY[14] §5 Estimating the Round-Trip Time.

5.2 Congestion Control

Decoupling of congestion control from reliability control: QUIC uses packet numbers for congestion control, and stream frame offset for reliability control.

Incorporating existing algorithms: Similar to TCP congestion control, QUIC utilizes a window-based congestion control scheme that limits the maximum number of bytes the sender might have in transit at any time. QUIC does not aim to develop its own new congestion control *algorithms*, nor use any specific one (e.g., Cubic). QUIC provides generic signals for congestion control, and the sender is free to implement its own congestion control mechanisms. A congestion control algorithm documented in the QUIC standard is described in appendix A.

To avoid unnecessary congestion window reduction, QUIC does not collapse the congestion window unless it detects *persistent congestion*. When two packets requiring acknowledgment are declared to be lost, persistent congestion will be established if none of the packets sent between them is acknowledged, an RTT sample existed before they were sent and the difference between their sent time exceeds the *persistent congestion duration* calculated based on the average RTT (`smoothed_rtt`), the deviation of the RTT samples (`rttvar`) and the maximum time the receiver might delay sending the acknowledgment.

A QUIC sender will pace its sending to reduce the chances of causing short-term congestion by ensuring its inter-packet sending interval exceeds a limit calculated based on the average RTT (`smoothed_rtt`), the congestion window size, and the packet size.

Related RFC: QUIC-RECOVERY[14] §7 Congestion Control.

5.3 Loss Detection and Recovery

ACK-based loss detection: As elaborated above, QUIC packets each contain several frames, each of which can be considered analogous to an IP packet. QUIC performs loss detection based on these packets (which is to say, the equivalent of a collection of IP packets): for each ACK'd packet, all frames carried in that packet are considered received. The frames carried in a packet are considered lost if that packet is unacknowledged when a later sent packet has been acknowledged, and when a certain threshold is met. QUIC uses two types of thresholds for determining whether an earlier sent packet is lost, (i) packet number based: the in-flight packet's sequence number is smaller than the acknowledged packet by a certain number. For instance, assuming the largest acknowledged packet number is x and the packet reordering threshold is t, then all in-flight packets with a packet number smaller than $x - t$ will be declared lost. (ii) time-based: the in-flight packet was sent at least certain times of the maximum of the current estimated network RTT and the latest sampled RTT before the acknowledged packet. For instance, assuming a packet was acknowledged at time t and the waiting time threshold is t_0, then all in-flight packets sent before time $t - t_0$ will be declared lost. These thresholds provide some grace period for packet reordering and avoid unnecessary retransmissions. It also aims to avoid performance degradation caused by the congestion controller when detecting packet loss.

To detect the loss of tail packets , QUIC will initialize a timer for the Probe Timeout (PTO) period whenever a packet requiring acknowledgment is sent, which includes the estimated network RTT `smoothed_rtt`, the variation in the RTT samples `rttvar`, and the maximum time a receiver might delay sending an acknowledgment. When the PTO timer expires, the sender will send a new packet requiring acknowledgment as a probe, which could repeat some in-flight data to reduce the number of retransmissions.

Loss Recovery: After a loss has been detected, the lost frames are then put into new outgoing packets (which will be assigned new packet numbers, unrelated to the lost packets).

With loss detection and recovery, QUIC supports the reliable ordered byte-stream delivery similar to the functionality provided by TCP.

Related RFC: QUIC-RECOVERY[14] §6 Loss Detection.

6 QUIC SECURITY

QUIC makes security a first-class priority in the protocol. As a result, QUIC encrypts almost everything within the protocol (outside of fields necessary for the network, e.g., source and destination addresses), along with tightly integrating security. Part of this integration is combining the transport and security handshakes: this allows one to cut 1 RTT off from the connection setup time. Though this may naively seem to violate the engineering principle of separating out components, in the vast majority of cases where TCP is used, TLS is used on top of it, making this no sacrifice at all.

6.1 QUIC Cryptographic Handshake

QUIC uses keys derived from a TLS 1.3 handshake to protect the confidentiality and integrity of its packets [19] and the TLS handshake messages are carried in the CRYPTO frames in the Initial and Handshake packets which are coupled with the transport handshake process. The relationship between QUIC and TLS can be described as QUIC taking information from TLS (handshake messages, keys, etc), and in turn providing a reliable stream to TLS. The overall process for a 1-RTT certificate-based cryptographic handshake without client authentication is illustrated in Fig. 7 and proceeds as follows:

- The client initializes the cryptographic handshake process by sending the TLS ClientHello message in its Initial packet, which included the client's supported cipher suites, the public share of its ephemeral Diffie-Helman key, and its QUIC transport parameters. Other TLS extensions like server name indication can also be included in this ClientHello message.
- After the server received the client's Initial packet, it will reply with an Initial packet containing the TLS

ServerHello message, which includes the server's chosen cipher suite for the connection, the public share of the server's ephemeral Diffie-Helman key, and possibly other TLS extensions necessary to establish the cryptographic context. At this point, the server can derive the Master Secret with its private key and the client's public key and will use keys derived from the secret to protect all future packets.
The server will then send a Handshake packet containing the following TLS messages: the EncryptedExtensions message containing the server's QUIC transport parameters and other TLS extensions that are not required to establish the cryptographic context, the Certificate message containing the server's certificate chain, the Certificate Verify message proving the ownership of the certificate's private key, and the Finished message providing authentication of the handshake and the computed keys. [17]
The server can start sending application data with 1-RTT packets at this point though the liveness of the client has not been verified.
- After the client receives the server's Initial packet and Handshake packet, it can also derive the Master Secret of the connection and calculate the keys to protect all future packets. It will send the TLS Finished message in a Handshake packet to confirm the handshake. It can also start sending application data to the server with 1-RTT packets.
- After the receipt of the Finished message from the client, the server will send a HANDSHAKE_DONE frame in a 1-RTT packet to confirm the handshake being finished.

Because multiple QUIC packets can be encapsulated into a single UDP datagram, the above handshake process can be finished with only four UDP datagrams.

Sending 0-RTT data QUIC allows a client to send encrypted application data before the handshake is completed by reusing the negotiated parameters from a previous connection. [19] After a connection is established, a QUIC server can issue a pre-shared key (PSK) identity associated with the connection's resumption secret through a TLS NewSessionTicket message with a special maximum early data size to indicate that it will accept 0-RTT data. A QUIC client can remember that PSK identity and its associated secret along with other critical connection parameters so that it can go through the simplified 0-RTT handshake process when connecting to the server next time. The overall process of a 0-RTT handshake is illustrated in Fig. 8 and proceeds as follows:

- The client initializes the handshake process by sending an Initial packet similar to that in the 1-RTT handshake process with a few extra TLS extensions. The client will use the pre_shared_key extension to tell the server the PSK identity it has and use the early_data extension to signal that it has 0-RTT data to send. The application data is included in an 0-RTT packet protected with the resumption secret, which can be encapsulated into a single UDP datagram along with the Initial packet.
- The Server then uses the TLS stack to check validity (e.g. if the correct cipher-suite was used). If the packet passes this check, the server uses both the QUIC stack and the application protocol to check the packet's validity. Part of the QUIC stack check is ensuring that some additional transport state is associated with the session ticket, above and beyond the TLS 1.3 requirements.
If the server chooses not to accept the 0-RTT data, it will fall back to the 1-RTT handshake process and no special actions will be needed.
- The server will send an Initial packet and a Handshake packet similar to the 1-RTT handshake with a few changes. In the ServerHello message, the server will include the pre_shared_key extension to indicate that the PSK identity is accepted. In the Handshake packet, the TLS extensions early_data will be added to the EncryptedExtensions message to signal that the 0-RTT data is accepted, and the server does not need to send the Certificate and Certificate Verify message because its identity has already been verified. To provide forward secrecy for the new connection, the server will migrate to a new Master Secret combining both the old resumption secret and the shared secret derived from the new DH key exchange. The server will also send an ACK frame to acknowledge the 0-RTT packet.
- The client will migrate to the new Master Secret after receiving the public share of the server's ephemeral Diffie-Helman key. The remaining process is the same as the 1-RTT handshake.

There are, however, some additional best practices that should be followed. For instance, when using the 0-RTT mode, best practice dictates that the 0-RTT keys should only be used to protect data that is idempotent because 0-RTT packets are not protected against replay attacks. Additionally, in order to reduce security vulnerabilities, when a Client's cached information expires, the Server should reject the 0-RTT connection and send its authorization info as in the first time connection setup. This is built into the QUIC protocol, and the Client should have no trouble moving directly into a 'first time' connection setup.

Related RFC: The Transport Layer Security (TLS) Protocol Version 1.3 [17] §2 Protocol Overview and §4 Handshake

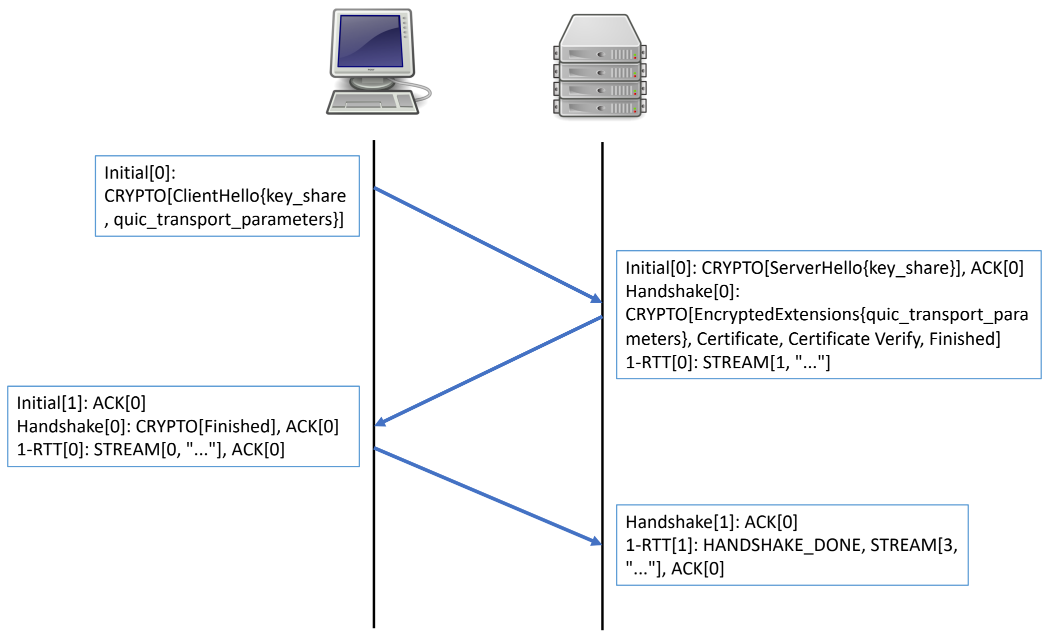
Initial[0]:
CRYPTO[ClientHello{key_share
, quic_transport_parameters}]
Initial[0]: CRYPTO[ServerHello{key_share}], ACK[0]
Handshake[0]:
CRYPTO[EncryptedExtensions{quic_transport_para
meters}, Certificate, Certificate Verify, Finished]
1-RTT[0]: STREAM[1, "..."]
Initial[1]: ACK[0]
Handshake[0]: CRYPTO[Finished], ACK[0]
1-RTT[0]: STREAM[0, "..."], ACK[0]
Handshake[1]: ACK[0]
1-RTT[1]: HANDSHAKE_DONE, STREAM[3,
"..."], ACK[0]

Figure 7: A diagram illustrating the QUIC-TLS 1.3 1-RTT handshake [15].

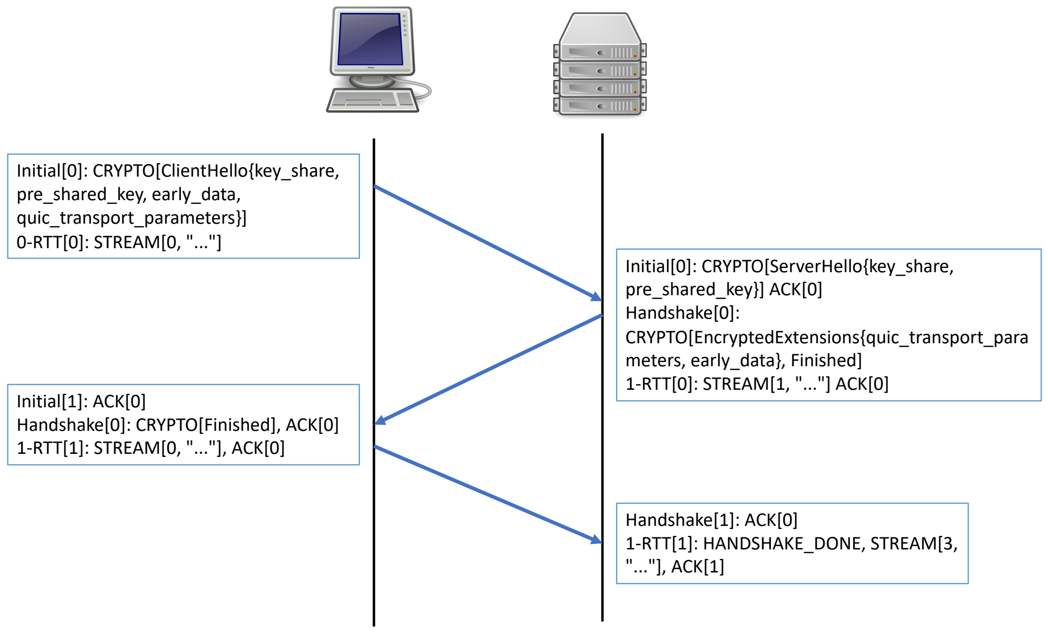
Initial[0]: CRYPTO[ClientHello{key_share,
pre_shared_key, early_data,
quic_transport_parameters}]
0-RTT[0]: STREAM[0, "..."]
Initial[0]: CRYPTO[ServerHello{key_share,
pre_shared_key}] ACK[0]
Handshake[0]:
CRYPTO[EncryptedExtensions{quic_transport_para
meters, early_data}, Finished]
1-RTT[0]: STREAM[1, "..."] ACK[0]
Initial[1]: ACK[0]
Handshake[0]: CRYPTO[Finished], ACK[0]
1-RTT[1]: STREAM[0, "..."], ACK[0]
Handshake[1]: ACK[0]
1-RTT[1]: HANDSHAKE_DONE, STREAM[3,
"..."], ACK[1]

Figure 8: A diagram illustrating the QUIC-TLS 1.3 0-RTT handshake [15].

Protocol, QUIC-TLS [19] §4 Carrying TLS Messages and §8 QUIC-Specific Adjustments to the TLS Handshake.

6.2 Authenticated and Encrypted Header and Payload

The QUIC protocol has made an intentional choice to encrypt all practical portions of the packet. Though this is a tradeoff – to give a specific example, hiding information from the ISP has both benefits and costs – it enables new protections for end users. To be more specific, QUIC authenticates all of its headers and payload (except version negotiation packets), as well as encrypting most of the data exchanged. Figure 4 shows the QUIC packet format, packet header fields are unencrypted as they are used for routing or performing decryption of the payload. The packet body containing frames is encrypted. Everything in the unencrypted header must remain in plaintext for proper operation. The header contains flags that are needed to specify which fields are included in the header and the length of certain fields. The Connection ID is used for routing the packet to its destination server and simultaneously serves as an identifier for the connection state. The packet number is needed for authentication and decryption, and thus can't be encrypted.

This encryption also has the benefit of ensuring QUIC can remain relatively easy to update. protocol ossification is a well-known problem – middleboxes cannot be easily upgraded to meet protocol changes, which limits the flexibility of network protocol design. QUIC packets are mostly encrypted, which prevents modification by middleboxes, and limits protocol ossification.

Related RFC: QUIC-TLS [19] §5 Packet Protection.

7 APPLICATIONS OVER QUIC: HTTP/3 AS AN EXAMPLE

As an application protocol, HTTP encodes application contents with rich semantics, and the most relevant feature to transport delivery is its request-response message exchange model: a browser client can issue multiple requests in parallel, whose responses often desire different delivery priorities to maximize the viewer experience. However, when HTTP runs over a TCP connection, which supports reliable delivery of a single byte steam only, the order of response contents can only be delivered according to the order in which the server receives the requests.

The QUIC design provides multiple stream support to address the above limitation of TCP being unable to prioritize between different requests. More specifically, HTTP/3, the latest version of HTTP which is designed to run over QUIC, utilizes QUIC's stream semantics to enable each HTTP response being delivered independently and with different priorities."

- As the underlying transport for HTTP/3, QUIC provides reliable in-order delivery at the stream level and congestion control at the connection level. Each HTTP request-response pair is mapped into an independent stream, thus different request-response pairs will not block each other in case of loss. QUIC also provides security matching TLS + TCP, and lower connection setup latency.
- Stream management is handled at the transport layer, QUIC takes care of the reliable delivery and the ordering of the frames then passes the received data to the application.

Related RFC: QUIC-HTTP[2] §4 HTTP Request Lifecycle, §6 Stream Mapping and Usage, and §7 HTTP Framing Layer.

8 CONCLUSION AND FUTURE WORK

QUIC represents the best transport protocol design the community has come out with so far. The basic ideas in the QUIC design did not drop out of the sky one day, but rather, QUIC represents an accumulation of lessons learned from networking experimentation and previous protocol designs over the last few decades. For instance, learning from T/TCP, QUIC keeps and reuses connection states to achieve 0-RTT communication. Adopting the ideas from RTP, QUIC runs over UDP to stay outside of the kernel and utilizes the ALF/ADU idea documented in [7]. Similar to SCTP and HTTP/2, QUIC also uses multiple substreams to mitigate head-of-line blocking and typed frames to support a variety of control exchanges. Adopting these ideas and synthesizing them into a single protocol allow the QUIC protocol to minimize latency and minimize other problems (such as those identified in §2.2).

- transport protocols to support reliable delivery to multiple parties.
- transport protocols to support delay tolerance.

REFERENCES

[1] Transmission Control Protocol. RFC 793, September 1981.

[2] Mike Bishop. Hypertext transfer protocol version 3 (http/3). Rfc, February 2021.

[3] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. TCP Congestion Control. RFC 5681, September 2009.

[4] Robert Braden. T/tcp – tcp extensions for transactions functional specification. RFC 1644, July 1994.

[5] Michelle Cotton, Lars Eggert, Dr. Joseph D. Touch, Magnus Westerlund, and Stuart Cheshire. Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry. RFC 6335, August 2011.

[6] Adam Langley et. al. The quic transport protocol: Design and internet-scale deployment. In *Proc. of SIGCOMM*, 2017.

[7] David Clark et. al. Architectural considerations for a new generation of protocols. In *Proc. of SIGCOMM*, 1990.

[8] Eric Rescorla et. al. Datagram transport layer security version 1.2. RFC 6347, January 2012.

[9] Henning Schulzrinne et. al. Rtp: A transport protocol for real-time applications. RFC 3550, July 2003.

[10] Tommy Pault et. al. An unreliable datagram extension to quic. Rfc, March 2021.

[11] Bryan Ford. Structured streams: a new transport abstraction. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 361–372, 2007.

```
              New Path or      +------------+
         persistent congestion |    Slow    |
     (O)---------------------->|    Start   |
                               +------------+
                                      |
                              Loss or |
                      ECN-CE increase |
                                      v
+------------+     Loss or     +------------+
| Congestion |  ECN-CE increase |  Recovery  |
| Avoidance  |----------------->|   Period   |
+------------+                 +------------+
          ^                           |
          |                           |
          +---------------------------+
             Acknowledgment of packet
               sent during recovery
```

Figure 9: State Machine of the New Reno Algorithm. [14]

[12] IANA. Service Name and Transport Protocol Port Number Registry. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml, 2021 (Last Updated 2021-10-04).

[13] Jana Iyengar. The maturing of quic, fastly, industry insights. https://www.fastly.com/blog/maturing-of-quic, November 2019.

[14] Jana Iyengar and Ian Swett. QUIC Loss Detection and Congestion Control. RFC 9002, May 2021.

[15] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

[16] Jon Postel. User datagram protocol. RFC 768, August 1980.

[17] Eric Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, August 2018.

[18] Randall Stewart. Stream control transmission protocol. RFC 4960, September 2007.

[19] Martin Thomson and Sean Turner. Using TLS to Secure QUIC. RFC 9001, May 2021.

A THE NEWRENO ALGORITHM

The state machine of the *NewReno* algorithm documented in the QUIC standard [14] is shown in Figure 9. In addition to the congestion window, the *NewReno* algorithm will also maintain another variable named Slow Start Threshold, which will be initialized to infinity. The *NewReno* algorithm has the following three states:

Slow Start: A QUIC sender will start at the Slow Start state and will reenter it when persistent congestion is declared. During this state, the congestion window will grow exponentially and is increased by the number of newly acknowledged bytes. The sender will enter the Recovery state when a packet is declared lost or when the ECN-CE counter has been increased.

Recovery: Each time the sender enters the Recovery state, the congestion window will be reduced by half and the Slow Start Threshold will be set to the new congestion window size. The sender will enter the Congestion Avoidance state when a packet sent during the Recovery state is acknowledged by its peer.

Congestion Avoidance: During this state, the Additive Increase Multiplicative Decrease (AIMD) approach will be used, and for each congestion window acknowledgment, the increase of the window size will be limited to the maximum size of one datagram. The sender will enter the Recovery state when a packet is declared lost or when the ECN-CE counter has been increased.

Handling Persistent Congestion: When persistent congestion is declared, the congestion window will be reduced to the minimum congestion window size and the sender will reenter the Slow Start state.

Related RFC: QUIC-RECOVERY[14] §7 Congestion Control.

B VERSION NEGOTIATION

Unlike traditional transport protocols, QUIC supports the co-existence of different protocol versions. In order to carry out this feature, the client and server can negotiate a mutually supported protocol version before establishing a connection. This is useful for the protocol to continuously evolve while allowing endpoints to negotiate which version to use. For clients that support multiple QUIC versions, QUIC should choose the largest of the minimum packet sizes across all the supported versions as the size of its first packet. If the server does not accept the version selected by the client, it will send an additional Version Negotiation packet to the client listing its accepted versions. This will introduce an additional 1-RTT latency to the QUIC handshake process.

Related RFC: QUIC-TRANSPORT[15] §6 Version Negotiation.