Computer Networks - COEN 233

QUIC

Quick UDP Internet Connections

Puneet Kumar
SCU-ID: 00001424550

Santa Clara University
Computer Networks: COEN-233
Department of Computer Science
California, USA

Audience

This document is intended to explain the new transport protocol which is going to revolutionize the network industry. It is important to have prior knowledge of the transport protocols (TCP and UDP). QUIC is exploring it's way into TCP world on a fast pace. This documents explains the protocol in a great detail. It targets audience of the class who are taking taken COEN - 233 (Computer Networks), as fundamentals to understand this document was explained in the class, as in lecture slides, OSI layer, transport layer protocols and it's interaction with layer 3(Network layer) and layer 7 (Application layer) are covered.

Contents

Audience ii

1 Introduction 1

1.1 Connection Overhead 1

1.2 Migrating Flows 3

1.3 Multiple Connections to Same Server 4

1.4 TCP Half-Open Connection 4

1.5 Head-of-Line Blocking (HOL) 4

1.6 Contribution 5

2 QUIC Protocol 6

2.1 QUIC Features 7

2.1.1 Connection Establishment 7

2.1.2 Improved Congestion Control 8

2.1.3 Multiplexing 8

2.1.4 Forward Error Correction 9

2.1.5 Connection Migration 9

2.1.6 Built-in Security 9

2.1.7 Solves Head-of-Line Blocking 10

2.1.8 NAT Rebinding 10

2.1.9 QPACK 11

2.2 QUIC Packet Format 11

2.2.1 QUIC Header 11

2.2.2 Connection Identifier 12

2.2.3 Stream Frame 12

2.2.4 Packet Number 13

2.2.5 Flow Control 15

2.2.5.1 Data Flow Control 15

2.2.5.2 Flow Credits Increments 15
2.2.6 Re-transmission 16

3 Conclusion and Future Aspects 17
3.0.1 User internet traffic 17
3.0.2 IoT Traffic 17
3.0.3 Future Aspect 18

4 Acronyms 19

Bibliography 21

List of Figures

1.1 TCP Connection Establishment 1

1.2 Total number of RTTs 3

1.3 TCP Half-Open Connection 4

1.4 Head-of-Line Blocking 5

2.1 QUIC Protocol in Network Stack 6

2.2 1 RTT QUIC connection establishment 7

2.3 1 RTT QUIC connection establishment 8

2.4 TCP vs QUIC Connections 9

2.5 NAT Rebinding 10

2.6 The solid and dashed lines show the clear-text and encrypted parts of a QUIC packet, respectively. The non-encrypted part is used for routing and decrypting the encrypted part of the packet. 12

2.7 Inside of a QUIC Packet 14

2.8 ACK Frame 16

List of Tables

1.1 TCP Versions Comparison 2

2.1 Type of Long Header 12

3.1 Comparison of TCP, UDP and QUIC 18

1. Introduction

Main responsibility of a transport protocol is to support communication between two end-to-end entities. These entities can be hosts or devices, such as routers, firewalls etc. Transport protocol provides mechanism to have a virtually circuited route between these two end-to-end devices. Transport protocol are of two types: Connection oriented and connection-less. As these name suggest, connection oriented requires additional work of setting up connections and provide reliability in terms of packet re-ordering, congestion control, reliable delivery of packets. On the other hand connection-less transport protocol adapts, sends and forgets mechanism. It's entire purpose is to deliver the datagram without having to worry about it's delivery. There are two major protocols, TCP (Transmission Control Protocol) [14] and UDP (User Datagram Protocol) [13], used for connection oriented and connection-less services respectively. Being a connection oriented protocol, TCP provides a reliable end-to-end connection and have congestion control mechanisms to avoid buffer overflow at the receiver side.

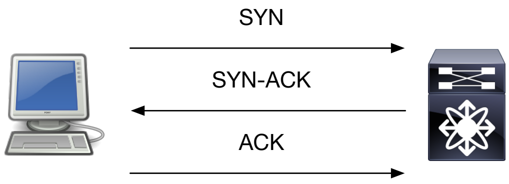


Figure 1.1: TCP Connection Establishment

1.1 Connection Overhead

Since TCP is the most important connection oriented protocol, it has been a center point of the research. Several improved version of TCP have been published in the last couple of decades(refer Table 1.1). These TCP versions have mainly focused on the throughput [5, 19]. Albeit, throughput is the main factor for determining the quality of traffic, TCP might not be for suitable for a lot of different types of traffic. Some of the examples of such traffic are: IoT (Internet of Things) [18], IIoT (Industrial Internet of Things) [17]. Traffic in these technologies are generally characterized by short-lived bursts of exchanging small data chunks. For such short lived connections, establishing connections every single time before an exchange of short live burst data is a lot of overhead. Connection start up latency in TCP is high, as it requires two round trips (RTT) for declaring a peer-to-peer connection alive and ready.

Table 1.1: TCP Versions Comparison

Version	Feedback	Required Changes	Benefits	Fairness
Reno (New)	Loss	–	–	Delay
Vegas	Delay	Sender	Less Loss	Propotional
High Speed	Loss	Sender	High Bandwidth	
BIC	Loss	Sender	High Bandwidth	
CUBIC	Loss	Sender	High Bandwidth	
C2TCP	Loss/Delay	Sender	Ultra-low latency high bandwidth	
NATCP	Multi-bit Signal	Sender	Near Optimal Performance	
Elastic-TCP	Loss and Delay	Sender	High Bandwidth/Short Long-distance	
Agile-TCP	Loss	Sender	High Bandwidth/Short Distance	
H-TCP	Loss	Sender	High Bandwidth	
FAST	Delay	Sender	High Bandwidth	Proportional
Compound TCP	Loss/Delay	Sender	High Bandwidth	Proportional
Westwood	Loss/Delay	Sender	L	
Jersey	Loss/Delay	Sender	L	
BBR	Delay	Sender	BLVC, Bufferbloat	
CLAMP	Multi-bit Signal	Receiver, Router	V	Max-Min
TFRC	Loss	Sender, Receiver	No Retransmission	Minimum Delay
XCP	Multi-bit Signal	Sender, Receiver, Router	BLFC	Max-Min
VCP	2-bit Signal	Sender, Receiver, Router	BLF	Proportional
MaxNet	Multi-bit Signal	Sender, Receiver, Router	BLFSC	Max-Min
JetMax	Multi-bit Signal	Sender, Receiver, Router	High Bandwidth	Max-Min
RED	Loss	Router	Reduced Delay	
ECN	Single-bit Signal	Sender, Receiver, Router	Reduced Loss	

Generally, applications which are required to have connection oriented transport protocol such as HTTPS [15], have some form of security enabled on top of transport layer. One of the most popular protocol used for such purpose is TLS (Transport Layer Security) [4]. This protocol provides security over the TCP by encrypting everything in application layer. Encryption is based on the keys exchange happened between end entities. This process adds one or two RTTs into the additional two RTTs in connection establishment between end points after connection establishment. Securing transport layer is necessary as it provides symmetric encryption which enables authenticated, confidential and integrity-preserved communication between two devices. TLS protocol has it's own handshake like TCP and it is unique per connection. So total 3-4 RTTs are required to establish and secure a connection oriented connection between end-to-end hosts.

As we have seen, it takes 3-4 RTT just to establish a TCP connection. Networks which are lossy in nature, such as wireless networks, will pay a heavy price of connection overhead. Everytime an IoT device gets disconnected from the network, to connect to the desired device again, TCP connection needs to be re-established. If a network has a lot of lossy connections, the amount of connection establishment overhead can be enormous.

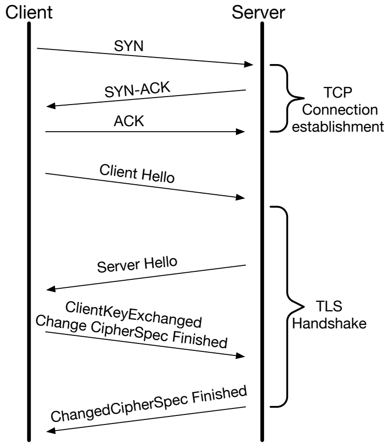

Client
Server
SYN
SYN-ACK
ACK
TCP Connection establishment
Client Hello
Server Hello
ClientKeyExchanged
Change CipherSpec Finished
TLS Handshake
ChangedCipherSpec Finished

Figure 1.2: Total number of RTTs

1.2 Migrating Flows

Limitations of TCP is not stopped only to connection overhead. Another major price TCP connection pays is flow migration. Flow is generally identified by a combination of several factors to identify single connection. Parameters involved in identifying a flow are source IP address, destination IP address, source port number, destination port number and the protocol (called 5-tuple). Some of these parameters (port numbers) are unique for a connection while others are common across device (ip address). Changing the application or device can lead to a different identifier than the previous one. This will eventually lead to tear down the establish TCP connection. Such condition will force devices to establish a new TCP connection and perhaps negotiate TLS handshake again. This could lead to a catastrophic event, imagine a user is paying a credit card bill and the money has been reduced from his account, but while contacting the vendor, connection is teared down. All session information prior to this connection is gone and now in new connection, packets have no information about prior bill pay. Second limitation arises due to flow migration is NAT rebinding. Generally, users are behind firewall in their LAN network, with non-routable IP (such as 192.168.x.x or 172.16.x.x), these IP and port makes a NAT table in the outgoing router to identify each device. Upon connection migration, this table can be inconsistent and can lead to breach in LAN hosts securities.

1.3 Multiple Connections to Same Server

Some applications, specially the web applications, such as HTTP [6] require to have multiple TCP connections established with the server in order to fetch data. These connection occupies bandwidth and bring connection overhead with them. Currently there is only one flavor of TCP out there called SPDY [1] which supports multiplexing (SCTP is not a flavor of TCP). Problem with SPDY is that it is not supported by many networks and it's limited usage to HTTP2 hindered it's popularity.

1.4 TCP Half-Open Connection

Another limitation of TCP is half open connections. Receiving data in TCP is passive. It means that dropped connections are only detected by the sender and receiver has no way of detecting them. In order to understand this problem, lets take an example. Suppose a connection is being established between client and server. If a server is failed to receive an `ACK` from client, this port is continued to be occupied and resources will only be released once the TCP state transitions to `Closed State` after timer expires.

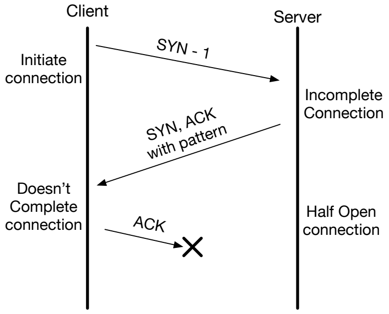


Figure 1.3: TCP Half-Open Connection

1.5 Head-of-Line Blocking (HOL)

Head-of-Line blocking problem is a limitation phenomena, which occurs when resources are being held up by some entity in order to complete an action. As shown in figure 1.4, three packets are coming from server and if packet 3 is dropped for some reason then packet 1 and 2 will not be pushed to application, until packet 3 is re-transmitted and packet re-ordering is completed. As one might guess, this phenomena only occurs if a connection oriented transport protocol is used. This out of delivery of packets can happen due to several reasons, such as lossy nature of the network, different paths taken by the packets and being dropped. HOL can significantly increase the packet reordering delay and which lead to consume resources and latency in packets being processed by network stack and application.

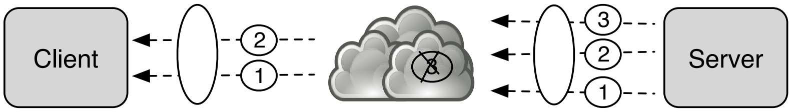


Figure 1.4: Head-of-Line Blocking

1.6 Contribution

With the increasing number of devices connected to internet, a new transport mechanism is needed to replace TCP to be able to address all the problems mentioned above. In this research report we will investigate and dissect QUIC [10] protocol. QUIC (Quick UDP Internet Connections) addresses these issues and bring a new perspective to the transport layer. Not only it provides fast transport but in terms of security it integrates TLS-1.3 which is the next generation transport layer security protocol. Chapter 2 will be focused on QUIC protocol specifically, chapter 3 will mainly talk about conclusion and explanation about the future aspects of the protocol.

2. QUIC Protocol

QUIC stands for Quick UDP Internet connections [10] and developed by Google. Initially this protocol was designed and developed for HTTP but later it was declared as general purpose transport layer protocol. This protocol provides security and reliability along with reduced connection and transport latency. Google has widely deployed QUIC in their servers and is currently in use.

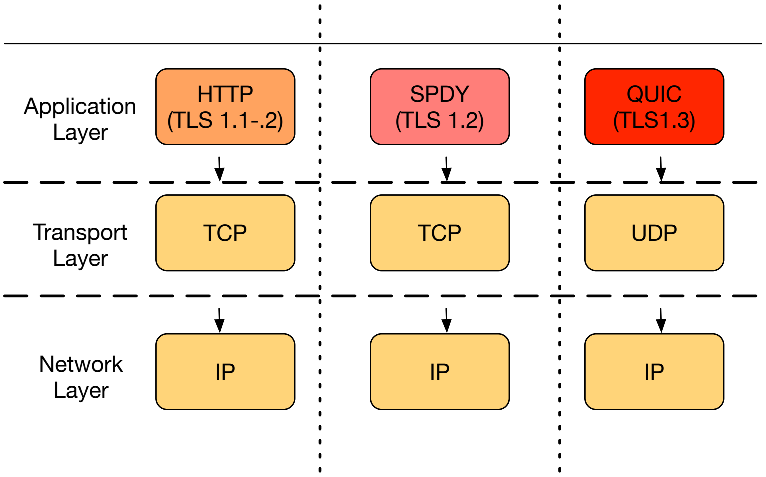


Figure 2.1: QUIC Protocol in Network Stack

As shown in figure 2.1, QUIC sits in application layer unlike other traditional transport layer protocol such as TCP or UDP. QUIC runs on UDP to be compatible with all the middleboxes. When middleboxes encounter QUIC traffic, they see only UDP traffic. The main idea behind QUIC is to overcome TCP limitations discussed in Chapter 1. As one might guess that the main transport protocol for QUIC, which is UDP, it is unreliable and doesn't provide any features needed for reliable connection.

QUIC solves the problems mentioned in Chapter 1. QUIC is very similar to TCP+TLS+HTTP2, but since it is implemented on top of UDP, all the connection reliability features are in application layer. As a self contained protocol, there are limitless opportunity for innovation in QUIC, which are not going to be possible with existing protocols as they bound to follow legacy.

2.1 QUIC Features

In this section, we will elaborate about the shortcomings we mentioned in Chapter 1. If we compare QUIC with TCP+TLS+HTTP2. It brings the following advantages:

- Connection establishment latency.
- Improved Congestion Control
- Multiplexing without head-of-line blocking
- Forward Error Correction
- Connection Migration

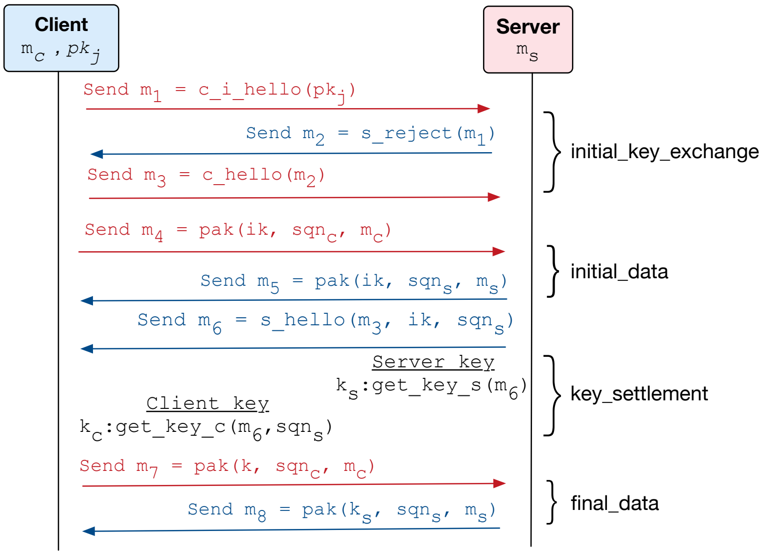

Client
m_c , pk_j
Server
m_s
Send m_1 = c_i_hello(pk_j)
Send m_2 = s_reject(m_1)
Send m_3 = c_hello(m_2)
initial_key_exchange
Send m_4 = pak(ik, sqn_c, m_c)
Send m_5 = pak(ik, sqn_s, m_s)
initial_data
Send m_6 = s_hello(m_3, ik, sqn_s)
Server key
k_s:get_key_s(m_6)
Client key
k_c:get_key_c(m_6,sqn_s)
key_settlement
Send m_7 = pak(k, sqn_c, m_c)
Send m_8 = pak(k_s, sqn_s, m_s)
final_data

Figure 2.2: 1 RTT QUIC connection establishment

2.1.1 Connection Establishment

As mentioned in chapter 1, TCP in conjunction with TLS, requires 3-4 RTTs to establish connection before data being encrypted and sent. This overhead makes applications slower and with the increasing demands of bandwidth and application availability, this overhead can be a bottleneck. QUIC solves this problem with having at most 1 RTT for fresh connection and 0 RTT for repetitive connections. If a connection is being established between client and server for the first time, then it must perform 1 RTT handshake in order to get the necessary information. This is done by client sending an empty `Client Hello (CHLO)` packet to server. Upon receiving a `CHLO` packet, server immediately send a `Rejection (REJ)` message to client. `REJ` contains information such as source address token, server's certificates etc. In order to move further in connection establishment, this information is necessary. Next time, the client sends a CHLO packet, which can use the cache credentials from the previous connection to immediately send encrypted requests to the server.

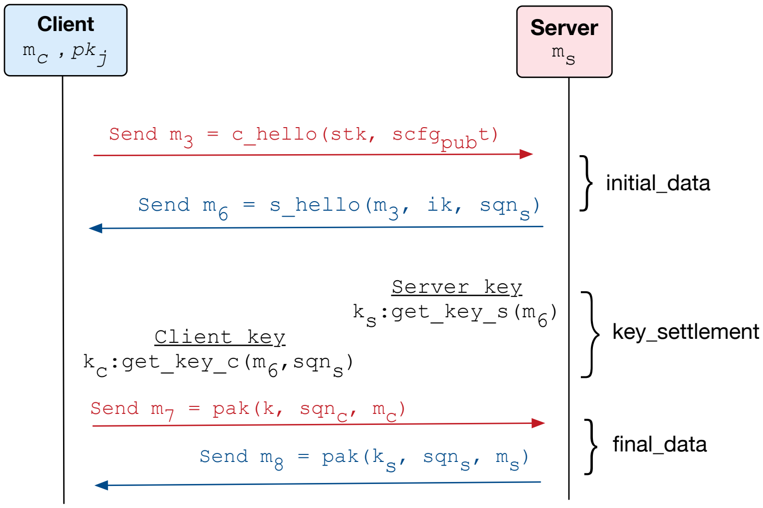

Client
m_c , pk_j
Server
m_s
Send m_3 = c_hello(stk, scfg_pub t)
Send m_6 = s_hello(m_3, ik, sqn_s)
initial_data
Server key
k_s:get_key_s(m_6)
Client key
k_c:get_key_c(m_6,sqn_s)
key_settlement
Send m_7 = pak(k, sqn_c, m_c)
Send m_8 = pak(k_s, sqn_s, m_s)
final_data

Figure 2.3: 1 RTT QUIC connection establishment

Figure 2.2 shows the packet exchanges in QUIC connection establishment in 1-RTT scenario. In 0-RTT scenario, server and client have communicated in the past and no need to establish connection again. As shown in figure 2.3, first packet itself carry the data, there is no need for connection establishment or key exchanges.

2.1.2 Improved Congestion Control

QUIC inherits TCP cubic algorithm for congestion control [8]. Unlike TCP, QUIC has pluggable congestion control, and provides richer information to the congestion control algorithm. For instance, each packet whether it is original or re-transmitted, has a unique sequence number. This simple hack provides QUIC sender to distinguish between original and re-transmitted packets and solves TCP ambiguity problem [7]. In addition to that QUIC ACKs also explicitly carry the delay between the acknowledgement sent and receipt of the packet. These sequence numbers are monotonically increasing numbers and do not get repeated in the connection lifetime.

2.1.3 Multiplexing

As mentioned in subsection 1.3, a client opens up multiple TCP connections to server to fetch data from server. QUIC solves this problem by having multiple streams over one UDP pipeline.

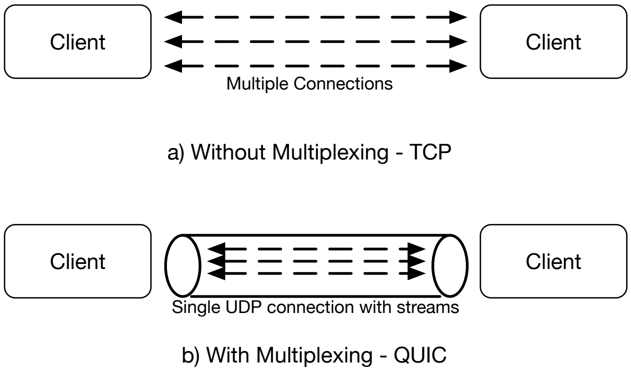

Client
Client
Multiple Connections
a) Without Multiplexing - TCP
Client
Client
Single UDP connection with streams
b) With Multiplexing - QUIC

Figure 2.4: TCP vs QUIC Connections

In QUIC, there is only one UDP connection for transport, while in the usage of TCP, there are multiple connections being established.

2.1.4 Forward Error Correction

Forward error correction mechanism is to recover the lost packets, without having to worry about re-transmission. QUIC can complement bunch of packets with an FEC packet. A perfect analogy would be RAID (Redundant Array of Independent Disks), just like RAID, FEC packet contains parity of the packets in the FEC group. If any packet is lost then content of the packet can be recovered from the FEC packet and the remaining packets in the group. It is up to the sender to decide whether to send the FEC packets for optimizing specific scenarios or not.

2.1.5 Connection Migration

As mentioned in section 1.2, TCP identifies flow by 5 tuple, connections in QUIC are being identified by a randomly generated 64 bit identifier called the `Connection ID`. In TCP, changing any parameter in 5-tuple can cause a connection break down and the session is no longer active. QUIC on the other hand, keep the `Connection ID` same and even if any parameter changes in 5-tuple underneath, it will not tear down the connection.

2.1.6 Built-in Security

In TCP world, choice of TLS depends on required security. It is an additional header gets imposed on top of TCP to encrypt the packet. This additional encryption comes with a cost of an extra handshake called TLS handshake. QUIC combines this handshake as a part of it's own handshake and do authentication of the end-points as well as negotiate all cryptography parameters. QUIC replaces TLS record layer with its own framing format, while keeping the same TLS handshake message. This ensures that connection is always authenticated and encrypted, but also in reduced RTTs.

In addition to that QUIC also encrypts additional metadata so that middleboxes such as firewalls, proxies do not manipulate the connection.

2.1.7 Solves Head-of-Line Blocking

QUIC uses UDP as a transport layer protocol, which doesn't make packets hostage before sending them to application. Network stack sees QUIC packets as datagrams and doesn't require to reorder them. QUIC receives those datagrams and if implementation allows, can be pushed to application in any order necessary. This makes sure that packets are received as soon as network stack pushes them to the application. QUIC also supports multiplexing which makes these packets in the same UDP pipeline and once it is established, no additional connection establishment overhead is required.

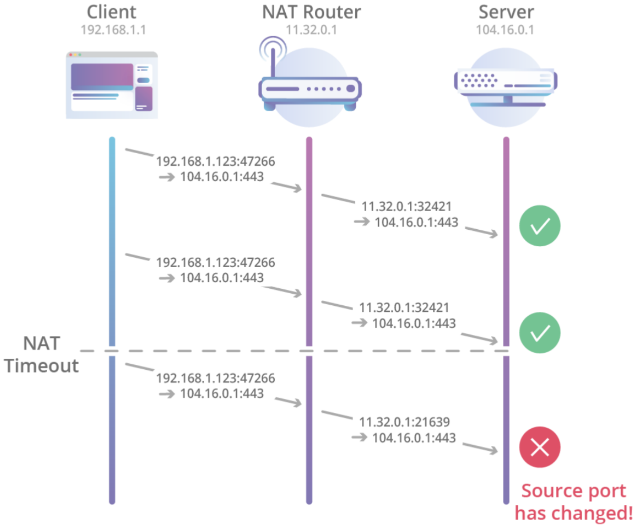


Figure 2.5: NAT Rebinding

2.1.8 NAT Rebinding

Typical NAT routers have a NAT table to keep track of all the TCP connections passing through them. The hash to identify the connection is calculated via 5-tuple and by observing SYN, ACK and FIN packets. It is easy to detect when a new connection is established, which it is done by looking at the TCP table (TCP state table). Having UDP as a transport protocol, NAT routers cannot have the state of a connection. NAT Routers will see this as UDP flows. When a NAT rebinding occurs, for instance, due to timeout, devices outside of the NAT perimeter will see those packets coming from different source port or IP than the one this flow was observed when the connection was originally established. This makes it impossible to track connections by only using the 5-tuple. Luckily, QUIC has connection ID which can be treated as hash of 5-tuple to identify the connection and the state can be determine by QUIC header (Described later).

2.1.9 QPACK

HTTP2 introduced header compression (HPACK)[12], which allows HTTP2 end hosts to reduce the amount of data transmitted. In particular HPACK has dynamic tables, filled by headers sent from previous HTTP requests/response. This allows end hosts to refer previously encountered headers in new requests, instead of having them transmitted again. HPACK tables need to be synchronized between HTTP sender and receiver. With TCP, synchronization is transparent, as TCP takes care of delivering HTTP requests and responses in the same order as they came in. On the other hand, QUIC can deliver multiple HTTP requests/responses, with the help of streams. This means that although it takes care of the data delivery in order, as long as it is a single stream, it doesn't guarantees any ordering among streams.

2.2 QUIC Packet Format

Figure 2.6 shows the general packet format for QUIC. Generally, there are a lot of variants of those packet formats, such as Long header, short header, version negotiation packet, initial packet, 1-RTT packet, 0-RTT packet etc. Discussing those packet format is out of scope for this project, but interested readers can go [9] to read about those packet structures.

2.2.1 QUIC Header

QUIC header is of two types: long header and short header. As name suggests, long header is big in size and short header is small. Long header contains more information and have the following fields:

- Version
- Destination Connection ID
- Source Connection ID
- Flags, such as header form, fixed bit, long packet type etc.

Table 2.1 shows the types of long header packets in QUIC. Long header are the packets sent prior to the establishment of 1-RTT keys (Initial packets). Once the keys are negotiated and connection establishment is completed, sender switches back to short header. Long header allows special packets such as version negotiation packets etc to be represented in uniform fixed-length packet format. On the other hand short packets are the most seen packets in QUIC traffic. Short header packets are used after the version and 1RTT keys negotiation. Short header packet contains the following:

- Destination Connection ID
- Packet Number
- Protected Payload

Table 2.1: Type of Long Header

Type	Name
0x0	Initial
0x1	0-RTT
0x2	Handshake
0x3	Retry

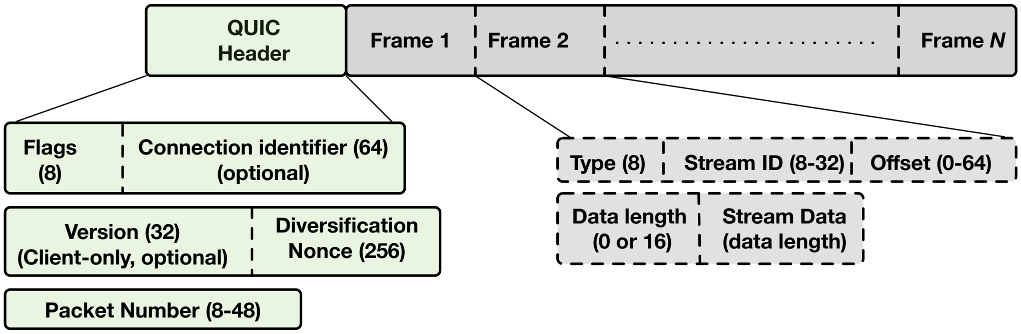

QUIC Header
Frame 1
Frame 2
Frame N
Flags (8)
Connection identifier (64) (optional)
Version (32) (Client-only, optional)
Diversification Nonce (256)
Packet Number (8-48)
Type (8)
Stream ID (8-32)
Offset (0-64)
Data length (0 or 16)
Stream Data (data length)

Figure 2.6: The solid and dashed lines show the clear-text and encrypted parts of a QUIC packet, respectively. The non-encrypted part is used for routing and decrypting the encrypted part of the packet.

2.2.2 Connection Identifier

Connection identifier is known as Connection ID or `CID`. Each connection has set of `CID`s, each of which can be used to distinguish a connection. `CID`s are independently selected by endpoints. The main task for `CID`s is to make sure that upon changing any lower payer protocols (UDP, IP) parameters, doesn't cause packets for a QUIC connection to be delivered to the wrong endpoint. Selecting `CID` is implementation specific, it is important because this method allows packets of that `CID` to be routed back to the endpoint and identified by the endpoint upon receipt. At any given point, `CID` should not contain any information that can cause the middleboxes to co-relate them with other `CID`s for the same connection. In other words, `CID` must not be issued again for a single connection. A zero-length `CID` can be used if a `CID` is not needed to route to the correct endpoint. Although, multiplexing connections on the same local IP address and port while using zero-length `CID` will cause failures in the presence of peer connection migration, NAT rebinding, and client port reuse. Hence, must not be done unless an endpoint is certain that those protocol features are not in use.

2.2.3 Stream Frame

As we have discussed before, QUIC is a multiplexed protocol. All streams are multiplexed over one UDP connection. Stream Frames implicitly creates stream and carry data. Each QUIC stream has a flow control and lost data transmitted mechanism. Which means that if data is lost on stream then it does not effect others.

Stream frames consists of the following:

- **Stream ID**: A variable length integer indicating the stream ID of the stream.
- **Offset**: A variable-length integer specifying the byte offset in the stream for the data in this stream frame.
- **Length**: An integer of variable length. It specifies the length of the stream data field.
- **Stream Data**: Bytes to be delivered.

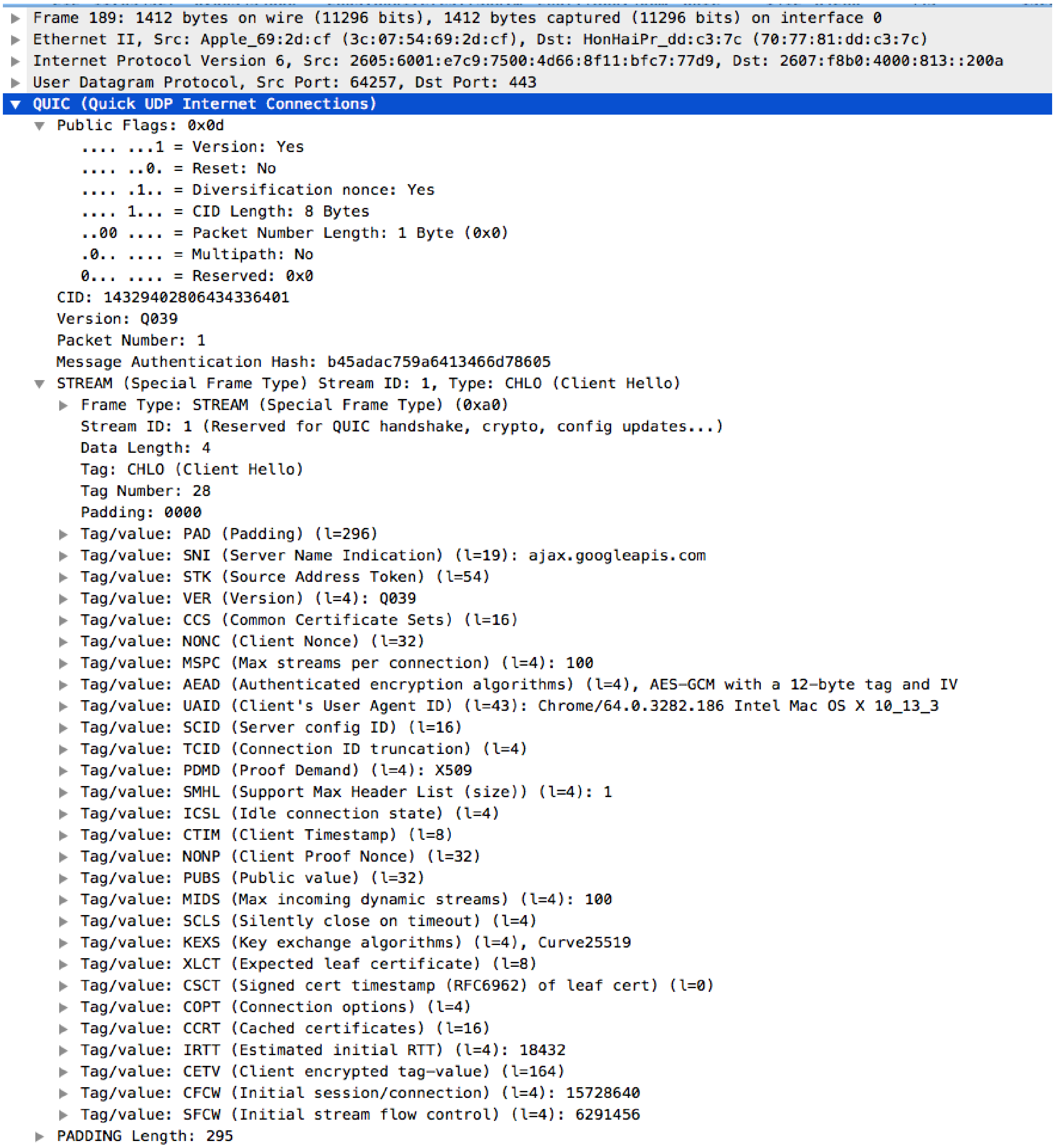
2.2.4 Packet Number

It is represented by an integer in the range of 0 to 2^{62-1}. It is used in determining the cryptographic nonce of the packet for protection. Each end points maintains a separate packet number for sending and receiving. Packet numbers are limited to this range because they need to be represented in whole in the largest acknowledged field of ACK frame [9]. Packet numbers can be reduced and encoded in 1-4 bytes. Packet numbers are divided into 3 spaces in QUIC:

- **Initial Space**: All initial packets are in this space.
- **Handshake Space**: All handshake packets are in this space.
- **Application Data Space**: All 0-RTT and 1-RTT encrypted packets are in this space.

A packet number space is a context in which a packet can be processed and acknowledged. For instance, initial packets can only be sent by initial packet protection keys. Similarly handshake packets are only sent at handshake encryption level. This brings cryptography separation between the data sent in the different packet sequence number spaces. Each space starts packets number as 0 and subsequent packets must increase the packet number by 1. 0-RTT and 1-RTT data exists in the same packet number space. There are certain additional rules to be followed in regards to packet number.

- A QUIC end point must not reuse packet number within same packet number space in one connection.
- If a packet number reaches 2^{62-1}, then the connection must be closed by sender without sending any notification.
- A newly unprotected packet must be discarded by receiver, unless receiver is certain that another packet bearing same packet number has not been processed.



```
▶ Frame 189: 1412 bytes on wire (11296 bits), 1412 bytes captured (11296 bits) on interface 0
▶ Ethernet II, Src: Apple_69:2d:cf (3c:07:54:69:2d:cf), Dst: HonHaiPr_dd:c3:7c (70:77:81:dd:c3:7c)
▶ Internet Protocol Version 6, Src: 2605:6001:e7c9:7500:4d66:8f11:bfc7:77d9, Dst: 2607:f8b0:4000:813::200a
▶ User Datagram Protocol, Src Port: 64257, Dst Port: 443
▼ QUIC (Quick UDP Internet Connections)
  ▼ Public Flags: 0x0d
      .... ...1 = Version: Yes
      .... ..0. = Reset: No
      .... .1.. = Diversification nonce: Yes
      .... 1... = CID Length: 8 Bytes
      ..00 .... = Packet Number Length: 1 Byte (0x0)
      .0.. .... = Multipath: No
      0... .... = Reserved: 0x0
    CID: 14329402806434336401
    Version: Q039
    Packet Number: 1
    Message Authentication Hash: b45adac759a6413466d78605
  ▼ STREAM (Special Frame Type) Stream ID: 1, Type: CHLO (Client Hello)
    ▶ Frame Type: STREAM (Special Frame Type) (0xa0)
      Stream ID: 1 (Reserved for QUIC handshake, crypto, config updates...)
      Data Length: 4
      Tag: CHLO (Client Hello)
      Tag Number: 28
      Padding: 0000
    ▶ Tag/value: PAD (Padding) (l=296)
    ▶ Tag/value: SNI (Server Name Indication) (l=19): ajax.googleapis.com
    ▶ Tag/value: STK (Source Address Token) (l=54)
    ▶ Tag/value: VER (Version) (l=4): Q039
    ▶ Tag/value: CCS (Common Certificate Sets) (l=16)
    ▶ Tag/value: NONC (Client Nonce) (l=32)
    ▶ Tag/value: MSPC (Max streams per connection) (l=4): 100
    ▶ Tag/value: AEAD (Authenticated encryption algorithms) (l=4), AES-GCM with a 12-byte tag and IV
    ▶ Tag/value: UAID (Client's User Agent ID) (l=43): Chrome/64.0.3282.186 Intel Mac OS X 10_13_3
    ▶ Tag/value: SCID (Server config ID) (l=16)
    ▶ Tag/value: TCID (Connection ID truncation) (l=4)
    ▶ Tag/value: PDMD (Proof Demand) (l=4): X509
    ▶ Tag/value: SMHL (Support Max Header List (size)) (l=4): 1
    ▶ Tag/value: ICSL (Idle connection state) (l=4)
    ▶ Tag/value: CTIM (Client Timestamp) (l=8)
    ▶ Tag/value: NONP (Client Proof Nonce) (l=32)
    ▶ Tag/value: PUBS (Public value) (l=32)
    ▶ Tag/value: MIDS (Max incoming dynamic streams) (l=4): 100
    ▶ Tag/value: SCLS (Silently close on timeout) (l=4)
    ▶ Tag/value: KEXS (Key exchange algorithms) (l=4), Curve25519
    ▶ Tag/value: XLCT (Expected leaf certificate) (l=8)
    ▶ Tag/value: CSCT (Signed cert timestamp (RFC6962) of leaf cert) (l=0)
    ▶ Tag/value: COPT (Connection options) (l=4)
    ▶ Tag/value: CCRT (Cached certificates) (l=16)
    ▶ Tag/value: IRTT (Estimated initial RTT) (l=4): 18432
    ▶ Tag/value: CETV (Client encrypted tag-value) (l=164)
    ▶ Tag/value: CFCW (Initial session/connection) (l=4): 15728640
    ▶ Tag/value: SFCW (Initial stream flow control) (l=4): 6291456
  ▶ PADDING Length: 295
```

Figure 2.7: Inside of a QUIC Packet

2.2.5 Flow Control

As any connection oriented protocol demands, there has to be a way to limit the amount of data a receiver could buffer. In order to impose that limit, flow control is necessary. TCP implements it's own way of flow control [16]. QUIC imposes flow control by controlling the maximum amount of data a sender can send on a stream at any given time. Also to limit concurrency within a connection, QUIC endpoint also controls the maximum cumulative number of streams that an endpoint can initiate. Since QUIC have TLS packets as a stream as shown in figure 2.7. Those streams are referred as `CRYPTO` frames. They are not flow controlled like stream data because QUIC relies to have an interface to the TLS about the buffer limit.

2.2.5.1 Data Flow Control

QUIC data flow control is similar to credit based flow control in HTTP2. It means that receiver advertises the number of bytes it is ready to received on a given stream per connection. Which leads it to two level of data flow control:

- **Stream Flow Control**: It prevents a single stream from consuming the entire buffer for a connection by limiting the amount of data that can be sent on any stream.
- **Connection Flow Control**: It prevents sender from exceeding a receiver's buffer capacity for the connection. It is done by limiting the total bytes of stream data sent in `STREAM` frames on all streams.

A receiver advertises credit for a stream by sending a `MAX_STREAM_DATA` frame with stream ID. `MAX_STREAM_DATA` frame indicates the maximum absolute byte offset of a stream. In addition to that, receiver also advertises credit for connection in `MAX_DATA` frame. This frame indicates the maximum of the sum of the absolute byte offsets of all streams. To close connection, receiver uses `FLOW_CONTROL_ERROR` frame, if sender sends any corrupted packet, with regards to advertised connection or stream data limits.

Sender on the other hand, must ignore any `MAX_STREAM_DATA` or `MAX_DATA` frames that do not increase any flow control limits. In case sender runs out of any credit, it will be unable to send any new data and will be considered blocked. In such event, a sender will send `STREAM_DATA_BLOCKED` or `DATA_BLOCKED` frame to indicate that it has data to write but is blocked by flow control limits.

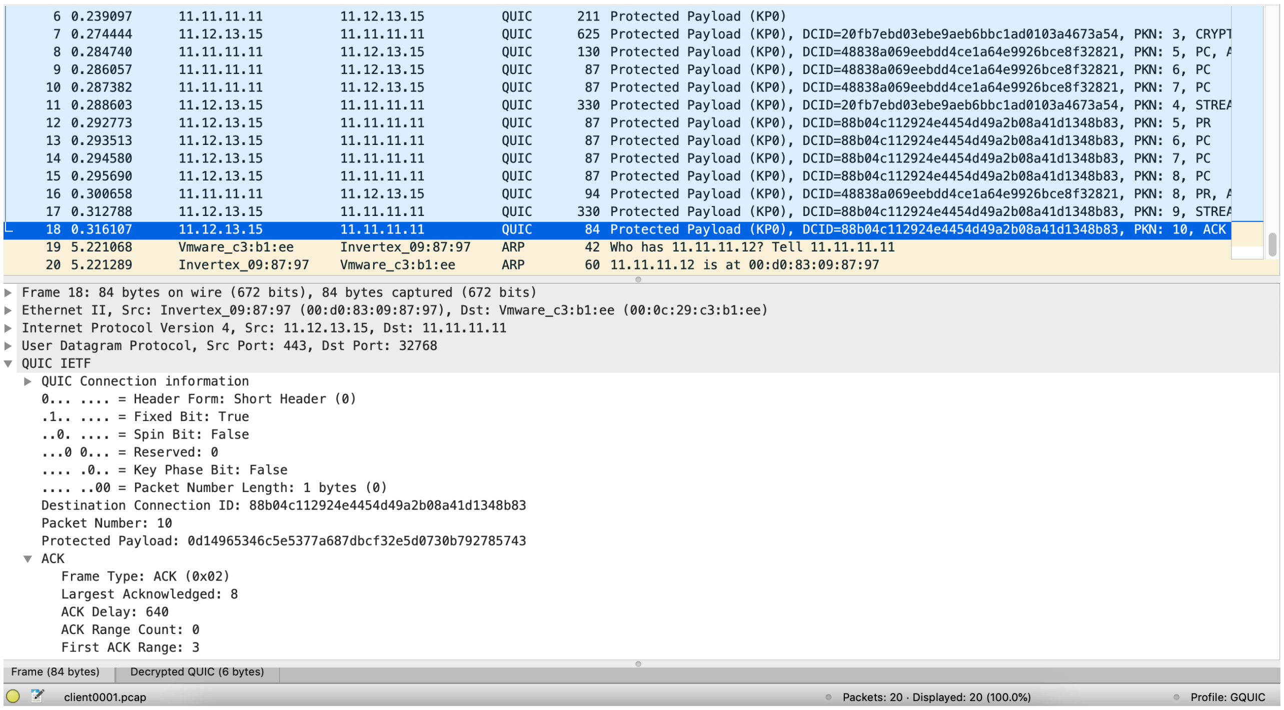
2.2.5.2 Flow Credits Increments

An auto tuning mechanism is used by receiver to tune the frequency and amount of the advertised additional credit based on the RTT time estimate and the rate at which the receiving application consumes data, just like what TCP does. As mentioned earlier, if sender runs out of credit and unable to send new data, it is considered blocked. To avoid it, and to reasonably account for the possibility of data loss, a receiver should send a `MAX_DATA` or

MAX_STREAM_DATA frame at least two RTT before it expects the sender to get blocked. In addition, a receiver must not wait for a STREAM_DATA_BLOCKED or DATA_BLOCKED frames before sending MAX_STREAM_DATA or MAX_DATA frame, since doing so will mean that a sender will be blocked for at least an entire rount trip, and potentially longer if the peer chooses not to send STREAM_DATA_BLOCKED or DATA_BLOCKED frame.

2.2.6 Re-transmission

In TCP world, packets are ACKed based on the sequence number and the next expected sequence number in the ACK packet. There are a lot of literature on TCP re-transmission, so we will not get into it. QUIC re-transmission is a bit different that TCP. As mentioned earlier, QUIC has multiple streams and any stream can be an ACK frame, acknowledging any past or present data frame. Due to this, QUIC maintains two counts, `ACK Range Count` and `First ACK Range`. `ACK Range Count` basically tells us till what packet number data frames has been received, and `First ACK Range` tells us till what packet number data frames have been received and acknowledged.



```
    6 0.239097      11.11.11.11        11.12.13.15        QUIC      211 Protected Payload (KP0)
    7 0.274444      11.12.13.15        11.11.11.11        QUIC      625 Protected Payload (KP0), DCID=20fb7ebd03ebe9aeb6bbc1ad0103a4673a54, PKN: 3, CRYPT
    8 0.284740      11.11.11.11        11.12.13.15        QUIC      130 Protected Payload (KP0), DCID=48838a069eebdd4ce1a64e9926bce8f32821, PKN: 5, PC, A
    9 0.286057      11.11.11.11        11.12.13.15        QUIC       87 Protected Payload (KP0), DCID=48838a069eebdd4ce1a64e9926bce8f32821, PKN: 6, PC
   10 0.287382      11.11.11.11        11.12.13.15        QUIC       87 Protected Payload (KP0), DCID=48838a069eebdd4ce1a64e9926bce8f32821, PKN: 7, PC
   11 0.288603      11.12.13.15        11.11.11.11        QUIC      330 Protected Payload (KP0), DCID=20fb7ebd03ebe9aeb6bbc1ad0103a4673a54, PKN: 4, STREA
   12 0.292773      11.12.13.15        11.11.11.11        QUIC       87 Protected Payload (KP0), DCID=88b04c112924e4454d49a2b08a41d1348b83, PKN: 5, PR
   13 0.293513      11.12.13.15        11.11.11.11        QUIC       87 Protected Payload (KP0), DCID=88b04c112924e4454d49a2b08a41d1348b83, PKN: 6, PC
   14 0.294580      11.12.13.15        11.11.11.11        QUIC       87 Protected Payload (KP0), DCID=88b04c112924e4454d49a2b08a41d1348b83, PKN: 7, PC
   15 0.295690      11.12.13.15        11.11.11.11        QUIC       87 Protected Payload (KP0), DCID=88b04c112924e4454d49a2b08a41d1348b83, PKN: 8, PC
   16 0.300658      11.11.11.11        11.12.13.15        QUIC       94 Protected Payload (KP0), DCID=48838a069eebdd4ce1a64e9926bce8f32821, PKN: 8, PR, A
   17 0.312788      11.12.13.15        11.11.11.11        QUIC      330 Protected Payload (KP0), DCID=88b04c112924e4454d49a2b08a41d1348b83, PKN: 9, STREA
   18 0.316107      11.12.13.15        11.11.11.11        QUIC       84 Protected Payload (KP0), DCID=88b04c112924e4454d49a2b08a41d1348b83, PKN: 10, ACK
   19 5.221068      Vmware_c3:b1:ee    Invertex_09:87:97  ARP        42 Who has 11.11.11.12? Tell 11.11.11.11
   20 5.221289      Invertex_09:87:97  Vmware_c3:b1:ee    ARP        60 11.11.11.12 is at 00:d0:83:09:87:97

▶ Frame 18: 84 bytes on wire (672 bits), 84 bytes captured (672 bits)
▶ Ethernet II, Src: Invertex_09:87:97 (00:d0:83:09:87:97), Dst: Vmware_c3:b1:ee (00:0c:29:c3:b1:ee)
▶ Internet Protocol Version 4, Src: 11.12.13.15, Dst: 11.11.11.11
▶ User Datagram Protocol, Src Port: 443, Dst Port: 32768
▼ QUIC IETF
  ▶ QUIC Connection information
    0... .... = Header Form: Short Header (0)
    .1.. .... = Fixed Bit: True
    ..0. .... = Spin Bit: False
    ...0 0... = Reserved: 0
    .... .0.. = Key Phase Bit: False
    .... ..00 = Packet Number Length: 1 bytes (0)
    Destination Connection ID: 88b04c112924e4454d49a2b08a41d1348b83
    Packet Number: 10
    Protected Payload: 0d14965346c5e5377a687dbcf32e5d0730b792785743
  ▼ ACK
      Frame Type: ACK (0x02)
      Largest Acknowledged: 8
      ACK Delay: 640
      ACK Range Count: 0
      First ACK Range: 3
```

Figure 2.8: ACK Frame

In case of any frame dropped, QUIC receiver tells sender via `Gap` in ACK frame. This tells sender how many packets were dropped and the NACK count tells what packet numbers are missing. Figure 2.8 shows re-transmission ACK frame.

3. Conclusion and Future Aspects

In this project, we examined different problems of TCP protocols faced today and explained their scenarios. In second chapter we dived down into QUIC protocol in detail. What we have found out is that QUIC protocol solves problems currently facing by TCP. Since QUIC sits in application layer, it brings substantial benefits, and it makes development and testing cycles fairly simple. QUIC is widely used in every type of traffic. Majorly it is used in user internet traffic and IoT traffic. We can conclude the QUIC protocol in these two parts.

3.0.1 User internet traffic

Major goal of QUIC was an experimental transport protocol. It has now become a core part of a serving infrastructure. It was thought initially that wide deployment of a new UDP-based encrypted transport for HTTP was an audacious goal; there were many unknowns, including whether UDP blocking or throttling would be show-stoppers. In our QUIC protocol analysis we showed that QUIC overcome the shortcoming imposed in user internet traffic over TCP. QUIC provides layering in terms of streams, which squashes away the HTTP layers in QUIC, which allowed to remove inefficiencies in the HTTP's stack. Finally the tussle between end hosts are inevitable, it can only be resolved when all interested parties come to the table [3]. Previously, attempts of deploying protocols that require any modification to network devices, such as SCTP [11], TCP Fast open [2]-name a few, have unequivocally exposed the difficulties of incentives and achieving consensus on proposed network changes.

3.0.2 IoT Traffic

IoT traffic is a bit different than user Internet traffic. Since IoT devices are usually deployed in lossy networks, the price they pay for a connection establishment while using TCP as a transport protocol is too much. QUIC mitigates that by introducing 0-RTT connection Establishment. QUIC eliminates the need of reconnecting upon changing any parameter in the 5-tuple. QUIC sees connection ID to identify a device and even if IP or port changes underneath, connection remains intact. QUIC solves head of line blocking problem by having streams mapped to one UDP pipeline. QUIC has better recovery mechanism by having FEC packets.

Table 3.1 shows the comparison between TCP, UDP, and QUIC.

Table 3.1: Comparison of TCP, UDP and QUIC

Solves	**TCP**	**UDP**	**QUIC**
Head-of-line Blocking	X	N/A	√
Multiple Connection to server	√	X	√
Connection Migration	X	X	√
Connection Overhead	X	X	√
Reliable Connection	√	X	√
Flow Control and Congestion Control	√	X	√

3.0.3 Future Aspect

Since QUIC sits in application layer, it utilizes CPU more than TCP. Modern NIC (Network Interface Card) has inbuilt TCP parameters offloading such as checksum and SSL offloading. QUIC is not there yet. CPU utilization in QUIC carries a heavy burden of encryption and decryption. QUIC also limits MTU (Maximum Transmission Unit) to 1392 bytes [9], which means that fragmentation is not yet supported in QUIC.

4. Acronyms

Below are the majorly used acronyms.

S.No.	**Acronym**	**Full Form**
1	TCP	Transmission Control Protocol
2.	UDP	User Datagram Protocol
3.	BIC	Binary Increase Congestion Control
4.	C2TCP	Cellular Controlled Delay TCP
5.	NATCP	Network Assisted TCP
6.	H-TCP	Hamilton Institue-TCP
7.	HTTPS	Hypertext Transfer Protocol Secure
8.	HTTP	Hypertext Transfer Protocol
9.	TLS	Transport Layer Security
10.	RTT	Round Trip Time
11.	IP	Internet Protocol
12	ACK	Acknowledgement
13.	SSL	Secure Socket Layer

Bibliography

[1] M Belshe and R Peon. Draft-mbelshe-httpbis-spdy-00: Spdy protocol, 2008.

[2] Yuchung Cheng, Jerry Chu, Sivasankar Radhakrishnan, and Arvind Jain. Tcp fast open. *RFC 7413 (Experimental), Internet Engineering Task Force*, 2014.

[3] David D Clark, John Wroclawski, Karen R Sollins, and Robert Braden. Tussle in cyberspace: defining tomorrow's internet. In *ACM SIGCOMM Computer Communication Review*, volume 32, pages 347–356. ACM, 2002.

[4] Tim Dierks and Eric Rescorla. Rfc 5246-the transport layer security (tls) protocol version 1.2. *Internet Engineering Task Force*, 2008.

[5] Sally Floyd, Andrei Gurtov, and Tom Henderson. The newreno modification to tcp's fast recovery algorithm. 2004.

[6] Network Working Group et al. Rfc 2616 hypertext transfer protocol–http/1.1. *online], http://www. faqs. org/rfcs/rfc2616. html*, 1999.

[7] Andrei Gurtov and Sally Floyd. Resolving acknowledgment ambiguity in non-sack tcp. *Next Generation Teletraffic and Wired/Wireless Advanced Networking (NEW2AN'04)*, 2004.

[8] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[9] ED Mozilla Ed Fastly J. Iyengar, M. Thomson. RFC Draft: QUIC: A UDP-Based Multiplexed and Secure Transport, 2019.

[10] Puneet Kumar and Behnam Dezfouli. Implementation and analysis of quic for mqtt. *Computer Networks*, 150:28–45, 2019.

[11] Shin Maruyama, Michael Tuexen, Randall Stewart, Qiaobing Xie, and Masahiro Kozuka. Stream control transmission protocol (sctp) dynamic address reconfiguration. 2007.

[12] Roberto Peon and Herve Ruellan. Hpack: Header compression for http/2. *Internet Requests for Comments, RFC Editor, RFC*, 7541, 2015.

[13] Jon Postel. User datagram protocol. *Isi*, 1980.

[14] Jon Postel. Transmission control protocol. 1981.

[15] Eric Rescorla. Rfc 2818: Http over tls. *Internet Engineering Task Force: http://www. ietf. org*, 2000.

[16] Timothy James Speight. Method and arrangement for tcp flow control, November 27 2012. US Patent 8,320,250.

[17] Jacob Wurm, Khoa Hoang, Orlando Arias, Ahmad-Reza Sadeghi, and Yier Jin. Security analysis on consumer and industrial iot devices. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 519–524. IEEE, 2016.

[18] Feng Xia, Laurence T Yang, Lizhe Wang, and Alexey Vinel. Internet of things. *International Journal of Communication Systems*, 25(9):1101, 2012.

[19] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control (bic) for fast long-distance networks. In *Ieee Infocom*, volume 4, pages 2514–2524. INSTITUTE OF ELECTRICAL ENGINEERS INC (IEEE), 2004.