QUIC Insight

Author: Jonas Reynders
Promoter: Prof. dr. Peter Quax
Co-Promoter: Prof. dr. Wim Lamotte
Supervisor: Mr. Robin Marx

2018

A thesis presented for the degree of Bachelor of Computer Science

Abstract

QUIC is a new transport protocol that is currently being developed by the QUIC working group. QUIC is build on top of UDP and will be integrating TLS and HTTP/2. However, this protocol will be designed from the ground up and existing features of TCP like congestion control, flow control, error handling, ..., will have to be redesigned. All of this makes the QUIC protocol complex and hard to understand. It is also difficult for developers to find issues within their implementations. There is a lack of existing tools that allow the QUIC protocol to be analysed. In this thesis we introduce a new tool for researching QUIC named QUIC-Vis. QUIC-Vis will make use of multiple resources of QUIC data to render different visualisations. These visualisations provide a solid experience for studying QUIC and debugging connection issues. More work can be done to improve the experience and to support more QUIC implementations.

Acknowledgements

First and foremost I would like to thank my mentor Robin Marx. From the beginning he explained the complexity of QUIC and what they are trying to achieve with designing a new protocol. His knowledge on QUIC was invaluable for me to get a basic understanding of the protocol. Robin also showcased that a tool such as Wireshark is not sufficient for studying QUIC, let alone using it to debug. His feedback on QUIC-Vis was invaluable in creating an application that researches would use to study QUIC.

I would also like to thank Kevin Pittevils. He is a developer of Quicker, which is one of the QUIC implementations. Kevin's knowledge of QUIC and Quicker was very helpful with configuring the Quicker client.

Finally, I would like to thank my promoter professor Peter Quax and my co-promoter professor Wim Lamotte for giving me the opportunity to write this thesis. The feedback of them and the Networking and Security research team were invaluable.

Contents

1 Introduction

The internet of today relies heavily on TCP. The first draft of this protocol grew from a protocol designed in 1974. Ever since then it has been in use, with adaptations being made along the way. Since the internet has evolved a lot over a relative short period, TCP has been mostly trying to catch up.

TCP became a quickly standard protocol for transmitting data on the internet. But over time it couldn't keep up with the ever-changing internet. Because of this middleboxes started appearing [7], which are pieces of software that "invade" the protocol. Middleboxes can improve performance, improve security, work around the shortage of IPv4 addresses, ... They work by looking directly at TCP packets and its options, sometimes even changing them. Further development of TCP is hindered by this, because there is no guarantee that a new option can go past the middleboxes. Introducing a change in TCP could take years, in order for middleboxes to handle this change.

To combat this, a proposal was made to create a new protocol. This protocol, SPDY, would run on top of TCP and would try improve TCP behaviour without the interference of middleboxes. At the same time a new version of the HTTP protocol would be designed alongside of SPDY. However, it was soon discovered that not all issues could be solved because of the problem existing in TCP itself. When HTTP/2 was finished, SPDY would be deprecated and HTTP/2 would run on TCP. There was still another transport protocol that could be used.

In 2012 a proposal was made by Google for a new transport protocol but this time designed on top of UDP (User Datagram Protocol). UDP is a very simple protocol which doesn't provide much extra of what the IP protocol does. So this new protocol called QUIC (Quick UDP Internet Connections) could have a fresh start and redesign the features TCP has like flow control, congestion control, in-order delivery, ... In 2016 a working group was formed to standardize QUIC. With HTTP/2 being available, QUIC could be designed to allow HTTP/2 make optimal use of its own features.

Designing a whole new transport protocol requires a lot of work. Deciding what the protocol has to do, how much of it does it need to do and how it needs to do it, are all important questions which need to be answered. That is why the QUIC working group are creating QUIC server/client implementations already. This is to make sure research can be done during development of QUIC, allowing researchers to discover any issues or improvements early on. For this, researchers have a need for tools to assist them with this research. With QUIC not even being standardized, not many tools are available.

The goal of this thesis was to develop a new QUIC debugging tool which provides the researchers and developers of QUIC, with a solid application they can use to analyse QUIC. The tool is called QUIC-Vis and will provide visualizations of the QUIC protocol. This will allow researchers to study and/or debug QUIC without spending a lot of time going through text-based tools.

In this work, we show that QUIC-vis can simplify research and debugging of the QUIC protocol. QUIC-Vis can both show high-level information of a QUIC connection in a structured way, but also low-level information that might be needed for debugging. We can also look at a connection in more detail and see how the server and client interact with each other. The tool still has room for more visualisations and reducing the amount of work the researcher has to do, to generate these visualisations.

2 State of the art

2.1 TCP and its problems

Currently TCP is being heavily used as a transport protocol. Though it is a very common protocol, it does not come without its own flaws. In this section we will discuss which major issues TCP has, some come from using TCP in combination with HTTP/2. Following this section, we will discuss ways to extend TCP and why there still needs to be a better alternative to TCP.

2.2 Latency

Let us start with looking at the way TCP sets up a connection [1]. First, the sender sends a SYN packet which will include connection options in its header. When the server receives that packet, it will reply with a SYN ACK. This packet will contain TCP flags and options. Once the sender receives the packet, it will respond with an ACK packet which confirms that the connection is set up. This handshake takes 1 round trip time (RTT) i.e., the sender does not wait for a reply to the last ACK packet before continuing.

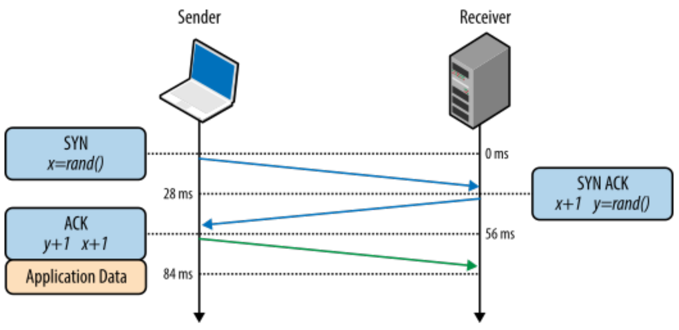


Figure 1: TCP 3-way handshake (https://hpbn.co/)

If the connection needs to encrypted, it has to be done during the connection set-up [1]. For securing a connection Transport Layer Security (TLS) is used. TLS is a separate protocols that runs on top of TCP and is responsible for encrypting data, authentication and checking integrity of data. Like with TCP, TLS needs some parameters to be enabled on a connection. This will extend the handshake in the beginning to 3 RTT.

The client first sends a ClientHello packet, that contains initial TLS options[1]. When the server receives it, it will reply with a ServerHello packet. With this packet the server will also send its certificate, which is used as way to authenticate. Once it is received by the client, it will initiate the key exchange for encryption. This will be sent to the server, which will process it. To this final key exchange message, the server will reply with a *finished* message, to indicate the connection has been set up.

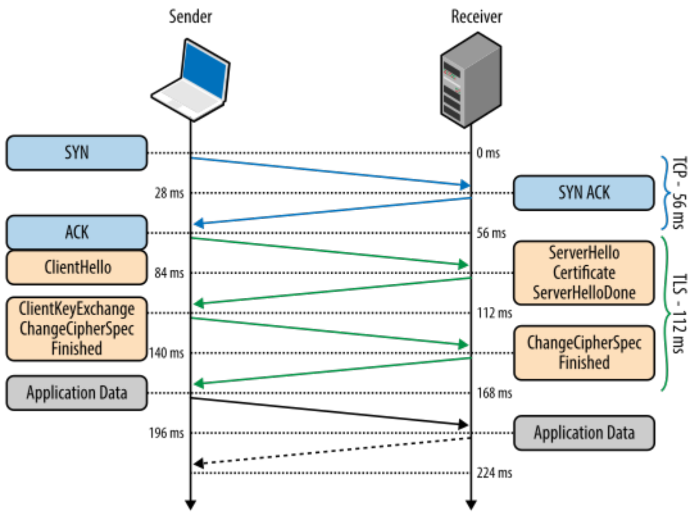


Figure 2: TCP + TLS handshake (https://hpbn.co/)

This part of the handshake will take an additional 2 RTT, on top of the initial 1 RTT from the TCP handshake. This adds a significant delay to sending data over a connection. For each connection set-up, this latency has to be taken in to account. The example shown in Figure 2 is a connection from New York to London on a fiber-optic cable [1]. There it takes around 168ms to set up a secure TCP connection with a RTT of 56ms. Another example could be a connection between New York and Sydney where the RTT is 160ms, it could take 480 ms to set up the connection.

2.3 Slow start

When a connection is opened, the client can start sending data. But, the client has no idea what the available bandwidth is between it and the server. It could send a certain amount of data at once, but that could immediately congest the network and cause packet loss. Congestion happens in a network when the incoming data exceeds the outgoing bandwidth. The network cannot process the packets fast enough so they start piling up. At first it will cause a delay for sending packets but when the buffer is filled up, new incoming packets cannot be stored and are then dropped. So what TCP does to prevent that, as well in order to reach an optimal, is called the slow start phase.

First, what TCP does in order to ensure data has arrived at the destination, is to use Acknowledgements[1]. This is a type of packet the receiver of data will send, so to inform the sender that the data has arrived and has not been lost along the way.

Then let us explain the concept of the congestion window (CWND). It is basically a buffer that contains data. All that data in the CWND is allowed to be sent over the connection without waiting on an acknowledgement from the server. When data is received at the other end and the client has received the acknowledgements, the data in the CWND can be replaced with other data that has to be sent.

When the handshake is done and data can be sent and an Initial Congestion Window(IW) is set. The size of the IW varies but is dependent on the Maximum Segment Size (MSS), which is the size of the largest amount of data that can be sent without being broken down in multiple packets. The size of the IW is a factor of the MSS, with factor 10 being the most common [10].

In the beginning only a handful of packets can be sent at once. Once those packets have arrived and acknowledgements have been received, then will the CWND grow in size. For each Ack recieved the CWND will grow in size by 1 Segment size. If it starts with an IW of 10 segments, after receiving 10 Acks the CWND will now be of size 20 segments. The CWND will keep growing until packet loss occurs, then its size will be cut in half and it starts by growing again but this time linearly [1].

For short data transfers, like HTTP requests, this adds an extra delay to sending a response. It could happen that an response is able to be sent in 1 RTT during a fully grown CWND, but during the slow start phase it may take 2 or more RTTs.

2.4 Congestion Control

The congestion control is present in TCP to help guarantee that all data is received at the other end. When packet loss occurs in the connection, congestion control goes into effect.

The growing and reducing of the CWND in Section 2.3 is present in TCP Reno. It is a loss based congestion control algorithm, but packet loss is not always tied to congestion. If part of the link has a small buffer, packet loss can occur because of a high amount of packets arriving. Even though that part of the link can send packet at the same rate they arrive. TCP BBR focusses on this issue, to respond to actual congestion and not just packet loss [1].

The downside of this can be seen when 1 TCP connection is used to send all data. Since it is the only connection that is running, its congestion window can grow large in size, which is nice because the transfer rate is going to be high. But, when there is packet loss, that CWND's size will be severely reduced. There is an advantage of using multiple TCP connections which have their own CWND. When packet loss occurs on one of the connections, only that connection's CWND will be changed[1].

As mentioned earlier in this chapter, there are multiple versions of TCP which have their own congestion control algorithm. However within each version of TCP, the congestion control algorithm can not easily be changed and applications can not really adapt to when congestion control is being active.

[1] https://cloudplatform.googleblog.com/2017/07/TCP-BBR-congestion-control-comes-to-GCP-your-Internet-just-got-faster.html

2.5 Head-of-Line blocking

HTTP/2 is the most recent version of HTTP at this time. Compared to the older versions, it has some additional features. One of those is the ability to define streams. It is basically a way to mirror having multiple connections, over one TCP connection, which solved the Head-of-Line blocking HTTP had [7]. Data of different streams could be sent concurrently.

Head-of-Line blocking occurs when multiple resources are being sent to the same end-point. Those resources does not rely on eachother and can be sent concurrently. The issue arises when packet loss happens on one of the resources. Even when a resource has been received, it has to wait before it will be delivered. When the lost packet(s) have been received, only then will all resources be delivered. As we can see in Figure 3[2], the first part of picture 1 has been lost during transmission. At same time we see that picture 2,3 and the second part of picture 1 has arrived. Picture 2 & 3 can be sent to the next layer and can be shown already, but they're being blocked because picture 1 is first in the order. Since picture 1 is not complete it needs to wait for a retransmission of the first part to arrive. Once that is done, picture 1 followed by 2 & 3 can be shown.

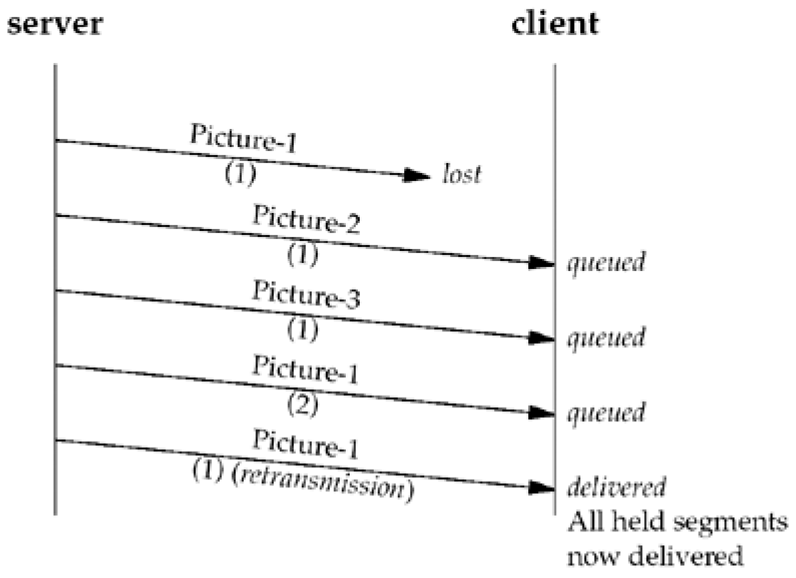


Figure 3: Head-of-Line blocking

The problem of HOL-blocking within HTTP may be fixed, but TCP has also a form of HOL-blocking. HOL-blocking within HTTP was caused by how requests for resources were handled. The order of requests being sent was also the order they would be processed. So the first resource needed to have arrived fully before the second resource would be processed.

[2] $http://www.masterraghu.com/subjects/np/introduction/unix_network_programming_v1.3/ch10lev1sec5.html$

HOL-blocking in TCP is caused by in-order delivery of all packets. So when a packet gets lost, all the data received from the subsequent packets will be delayed. Only when the lost packet is received, will all the received data be delivered to the application. Even when there are multiple resources which are independent of each other.

2.6 Connection migration

Devices can easily change from one network to another. A common example is wireless devices that move out of coverage of one network and into another. When this happens the device will have a new IP and the route to this device will be changed.

If this happens during a TCP connection, the original connection has to be closed and a new one has to be established. This is because a TCP endpoint is defined by defined by 2 IP/port combinations, called the 4-tuple [3]. When a client changes to another network and gets a new IP, it is basically a different endpoint. As we have seen previously, setting up a TCP connection will add a considerable amount of latency and require the buildup of the CWND which will delay the data transfer.

2.7 Header in plaintext

TCP headers are being sent over a connection in plaintext. Those headers are also not authenticated to an endpoint. This allows the TCP header to be altered/manipulated along the way to an endpoint. There are middleboxes that use certain techniques to improve TCP performance. Having those middleboxes present makes it harder to update TCP, since an update could cause those middleboxes to stop working correctly.

A middlebox can be used to improve the performance of TCP in a high bandwidth network [2]. TCP has the slow start in the beginning of a connection, which adds a delay till the data arrives. In order to reduce that delay and move past the slow start phase, a node in the network can send optimistic acks. Rather than waiting on the endpoint to sent it's own Acks, the node creates its own Acks which it sends to the other endpoint.

However, there also techniques that can impair TCP performance. Malicious attacks can be executed by middleboxes to intercept data or to do a Denial-of-Service attack. For example comparing to the situation above, there is the optimistic ACK attack. In this situation a middlebox will create ACK packets by itself to pretend to be the server. The client sees that all data is arriving and continues to grow the CWND. Which would congest the network.[6]

[3] $https://tools.ietf.org/html/rfc793$

2.8 Maintenance

The TCP protocol is a kernel based protocol since it is a protocol that is used very often and needs to perform fast. This introduces some difficulties in updating TCP. Let us assume TCP changes can be deployed, so those changes does not cause problems with middleboxes. Those changes have to still be pushed to a new kernel version. In other to get this new version, an OS upgrade has to happen.

An OS upgrade is going to have a system-wide effect, so it has to be rigorously tested to make sure it does not cause problems on the system. This will cause a big delay for deploying changes. Which is a danger when there are vulnerabilities present in the protocol.

2.9 Why QUIC

The internet has evolved in a relatively short period. There is more available bandwidth, a lot more services depend on the internet, the complexity of those services has increased, ... Over the years vulnerabilities are being discovered and protocols have evolved. TCP is still the protocol of choice for web applications, yet it has big issues that are hard or almost impossible to fix. There is HOL-blocking, amount of time it takes to set up a connection, the presence of middleboxes hindering updates to the protocol, ... There are plenty of proposal made to improve TCP: solving HOL-blocking [9], reducing latency when packet loss happens [9]. Given the current situation, with the presence of middle-boxes, causes complications. Changes in TCP can cause some of the middle-boxes to stop working, perhaps even block traffic. Waiting for those middle-boxes to catch up takes time, causing updates to TCP to take decades [7].

There was a new protocol in development which would focus on optimizing http traffic, called SPDY [11]. This is a protocol that runs on top of TCP and works in the application layer. After the first draft of the SDPY standard was published, the HTTP working group decided to use SPDY to develop a new version of HTTP: HTTP/2. With SPDY being build on top of TCP, it was soon discovered that the problems mentioned in this section would have a strong impact on SPDY. So when the HTTP/2 standard was finished, it was decided to deprecate SPDY. HTTP/2 now runs on top of TCP where it cannot run optimally because of the issues that TCP has, like HOL-blocking.

There is still an option to develop a new protocol in the application layer to optimize the transport layer. The User Datagram Protocol is another transport layer protocol which is standardized and deployed. It is however a very simple protocol which doesn't offer much compared to TCP like congestion/flow control, in-order delivery, handshake, ... So when designing a new protocol for transport that uses UDP, those features have to be implemented in that new protocol. It would require a lot of work but allows for many optimizations to be done in transporting data, improving on the problems TCP has. This is where QUIC comes in.

3 QUIC

3.1 What is QUIC

The QUIC protocol, or Quick UDP Internet Connections, is a new transport protocol originally designed by Google in 2012. It was also deployed at Google as an experiment. QUIC was proposed when issues with SPDY were discovered.

In 2016 the QUIC working group was formed to start working standardizing QUIC. They are also working closely together with the HTTPbis working group, mostly for the QUIC mapping for HTTP/2.

QUIC is a multiplexed transport protocol that has been built upon UDP, it also makes use of TLS 1.3 to secure connections. Since it is build on top of UDP, QUIC can be deployed as a user-level protocol. This will allow for more frequent updates and fixes. QUIC also provides better support and tools for application level protocols.

3.2 Core principles of QUIC

3.2.1 Reduce latency with handshake

In order to set up a reliable and secure connection some information has to be exchanged. So like with TCP, a handshake will be performed by QUIC. TCP's 3 RTT handshake, seen in Section 2.2, is necessary to set up a connection, so let us see how QUIC improves on that.
The way QUIC does this is by combining the transport handshake and the cryptographic handshake. The initial handshake will take 1 RTT, when it succeeds the client will cache certain information about the server it connected to. This information can be later used when connecting to the same server to do a 0 RTT handshake.

The initial 1-RTT handshake will happen when the client has no information about the server. So firstly, the client will send an $inchoateclienthello(CHLO)$ message to which the server will respond. This message will contain information about the server authentication and TLS parameters. Once the server config information has been received, the client verify it and will then send a $completeCHLO$. The client will also cache that information.

There is one requirement with making the handshake. The client's initial packet needs to be padded to reach a packet size of 1200 bytes. This is to ensure that the path of the connection supports a reasonable sized packet. The initial packet may be larger than 1200 in size as long as it is not larger than the Path Maximum Transmission Unit, which is the largest size a packet can be without needing to be fragmented in order to be sent.

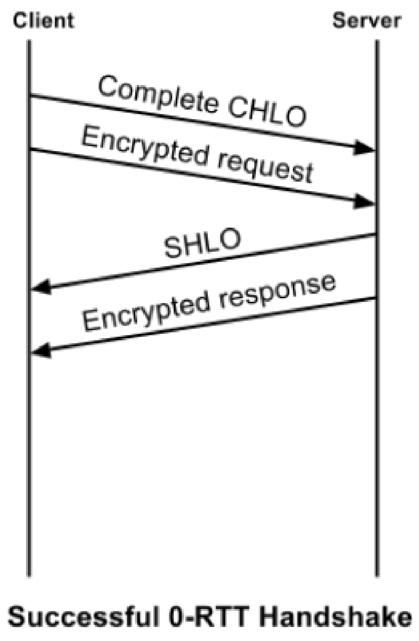


Figure 4: 0-RTT Handshake [7]

3.2.2 0-RTT handshake

An addition to the QUIC handshake is the ability to have a 0-RTT handshake. In order for this to be possible, the client needs to have connected to that server before with a 1-RTT handshake. The 0-RTT handshake (seen in Figure 4) consists of the $completeCHLO$ which after it is been sent, the client can already start sending data. It does not have to wait for a reply from the server. If the handshake is successful, the server will respond with a $serverhello(SHLO)$ message.

3.2.3 Multiplexing

QUIC also supports the multiplexing of data over multiple streams. This allows messages to be sent concurrently over a single connection. Compared to TCP (see Section 2.5), QUIC ensures that when using multiplexing no Head-of-Line blocking will occur on connection-level. Within each stream, the order of frames is still guaranteed and can still cause HOL-blocking. So when a packet is lost for a particular stream, the other streams will not have packets queued up until the lost data has been received. Packet order within a stream will still be respected when packet loss happens.

3.2.4 More feedback for congestion control

Firstly, QUIC will not rely on just one congestion control algorithm unlike each TCP version (Section 2.4). It will rather have a plug-in system in place to allow a congestion control algorithm of choice to be used. This is not only for choosing the right congestion control algorithm for a situation, but it also allows for new algorithms to easily be tested.

An interesting situation to look at is high latency networks where TCP doesn't perform well. An issue is the additive increase of the CWND, where after receiving an ACK the CWND grows with an linear amount [5]. Since the RTT is really high, it will take a while before TCP fully utilizes the available bandwidth.

Compared to TCP, QUIC packets will also carry more information related to congestion control and loss recovery. This will further increases the effectiveness of congestion control algorithm, since more data is available. Added in draft-13 was the inclusion of Explicit Congestion Notification (ECN). As the name suggests, it can signal when there is congestion in a network. Rather than dropping packets, packets are marked when congestion is present by routers in between the two endpoints. This is done on the IP layer, within its header and can only be done if all the nodes part of the connection have this enabled. QUIC can use this feedback by sending an ECN_ACK frame. This already helps to distinguish actual congestion from packet loss.

3.2.5 Authentication and encryption

QUIC packets will mostly be encrypted. The payload will always be and the header only parts of it will be encrypted compared to TCP (section 2.7). This will already help with stopping packet manipulation. As well as using encryption, all QUIC packets will be authenticated to further protect them from being manipulated by a third party. In order to protect the handshake packets from being tampered with, those packets will be verified when doing the cryptographic processing.

3.2.6 Connection migration

QUIC will make use of connection IDs to keep a connection alive when an endpoint changes to a different network or when NAT rebinding has occurred. Currently this is only allowed by the client. The client has to initiate the migration. It does that by sending probing frames to the server with a new IP address. The server has to then validate that it still is the same client, before sending additional data.

3.3 QUIC Packet

For the remainder of this section, we will be looking at the draft-11 version of QUIC [4]. We will be looking at draft-11 because it is currently the latest draft that Wireshark supports, which will be needed later for research purposes.

3.3.1 Header types

QUIC supports two header types: one is the long header and the other one is the short header. The long header is used for specific situations like during the connection set-up. It will be used to perform version negotiation and cryptographic handshake. Though the version negotiation packet uses an adaptation of the longer header. After the connection is set up, the client switches to the short header for further communication. This is to reduce the overhead for sending data.

Long header The long header contains the following fields:

- Header form: This bit will be set to 1 in order to indicate that a long header is used
- Long Packet Type: This value will inform the receiver the type of packet it is
- Version: This contains the version of QUIC that is currently being used, this will also determine how the rest of the fields are interpreted
- DCIL & SCIL: These will indicate the lengths of the SCID and DCID fields discussed below
- Destination Connection ID: This value will be used to identify the destination
- Source Connection ID: This value will be used to identify the source. There are two separate CIDs to represent 2 different paths: source -> dest and dest -> source
- Payload Length: An integer that represents the length of the payload field.
- Packet Number: This value will be used to distinguish different packets
- Payload: This will be the data that packet is carrying

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+
|1|   Type (7)  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Version (32)                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|DCIL(4)|SCIL(4)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Destination Connection ID (0/32..144)         ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 Source Connection ID (0/32..144)            ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Payload Length (i)                    ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Packet Number (32)                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Payload (*)                        ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

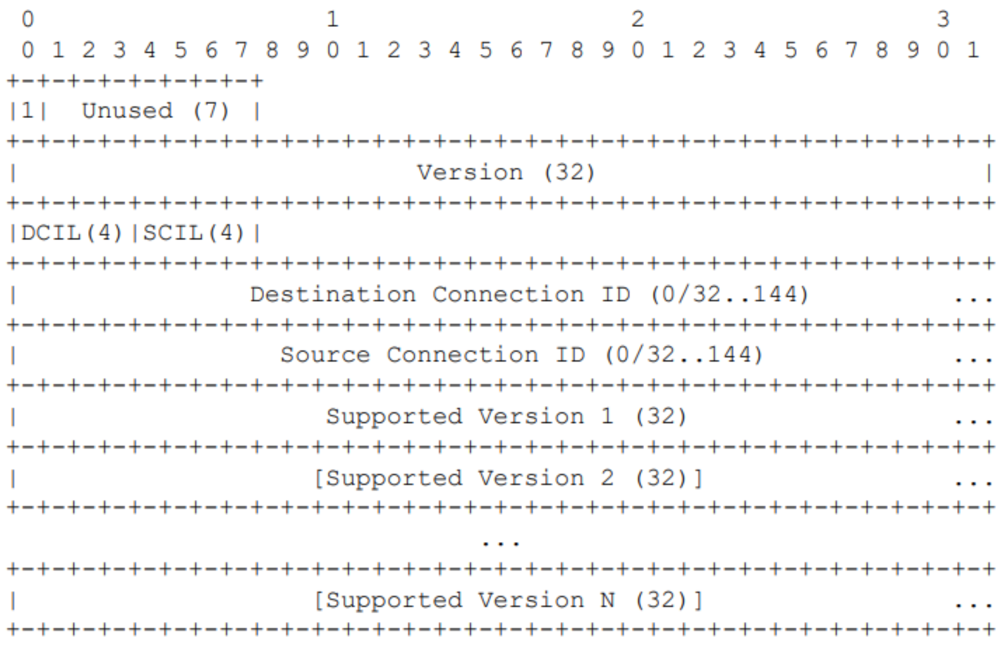
Figure 5: Long header [4]

Version Negotiation Header The version negotiation header contains the following fields:

- Header form: This bit will be set to 1 in order to indicate that a long header is used
- Unused: The value of this has no use during the version negotiation and will be randomly selected by the server
- Version: The total value of this will be 0 to indicate it is a version negotiation packet
- DCIL & SCIL: These will indicate the lengths of the SCID and DCID fields discussed below
- Destination Connection ID: This value will be used to identify the destination
- Source Connection ID: This value will be used to identify the source
- Supported Version X: The rest of the packet will contain a list of version that are supported by the server.

Short header The short header contains the following fields:

- Header form: This bit will be set to 0 in order to indicate that a short header is used



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+
|1|  Unused (7) |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Version (32)                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|DCIL(4)|SCIL(4)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Destination Connection ID (0/32..144)         ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 Source Connection ID (0/32..144)            ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Supported Version 1 (32)                 ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   [Supported Version 2 (32)]                ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                               ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   [Supported Version N (32)]                ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 6: Version negotiation header [4]

- Key Phase Bit: If this bit is toggled, it indicates that the keys used for protecting the packet have changed
- Reserved: This bit is reserved for experimentation.
- Short Packet Type: This value will inform the receiver the type of packet it is. Currently it only indicates how long the packet number is.
- Destination Connection ID: This value will be used to identify the receiver of the packet
- Packet Number: This value will be used to distinguish different packets. The length of the number will depend on the packet type that is defined.
- Protected Payload: This will be the data that packet is carrying, the data will be protected with cryptographic keys that have been defined prior in the connection

Stateless Reset A stateless reset has a special type of header (Figure 8) and is used for recovering from a severe connection error. It is used by the server if, for example it has crashed, cannot continue connection because it doesn't have access to its state. There are other ways to deal with errors which will be discussed later in this section.

The stateless reset header contains the following fields:

- Header form: This bit will be set to 0 in order to indicate that a short header is used

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+
|0|K|1|1|0|R|T T|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Destination Connection ID (0..144)           ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Packet Number (8/16/32)                ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Protected Payload (*)                   ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 7: Short header [4]

- Key Phase Bit: If this bit is toggled, it indicates that the keys used for protecting the packet have changed
- Short Packet Type: This value will inform the receiver the type of packet it is. Currently it only indicates how long the packet number is.
- Destination Connection ID: This will be a random generated CID.
- Packet Number: This will also be randomized.
- Random octets: This contains is a message with as length a random amount of octets. The message will consist of random values
- Stateless Reset Token: A token that only the client and server knows, it is for verifying that it is a stateless reset announced by the other endpoint.

The stateless reset token is generated by the server during the handshake. When a stateless reset happens, the client receives a stateless reset packet. This packet can either not be decrypted or the randomly generated packet number was already used in the connection. When this is the case, the client will check if the last 16 octets is equal to the stateless reset token that was given.

3.3.2 Frames

With the exception of Version Negotiation and Stateless Reset Packets, the payload of all packets with consist of frames. The Protected Payload will consist of at least 1 frame. A payload can also contain multiple frames, but the size of a frame or frames can not exceed the total packet size. There are multiple different types of frames with each having a different frame layout, though they all follow the generic frame structure (figure 9).

Being able to identify different types of frames allows each frame to have its own structure for the purpose it serves. For sending data, a Stream frame is used which simply carries binary data belonging to the same logical entity (e.g., one HTTP response, a single file).

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+
|0|K| Type (6)  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Destination Connection ID (144)              ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                   Packet Number (8/16/32)                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Random Octets (*)                       ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
+                                                               +
|                                                               |
+                  Stateless Reset Token (128)                  +
|                                                               |
+                                                               +
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 8: Stateless Reset [4]

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Type (8)    |           Type-Dependent Fields (*)         ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 9: Generic Frame Structure [4]

To acknowledge data, an Ack frame is used which only needs to focus on sending packet numbers that have arrived. Each frame having a different structure allows frames to be processed more easily than if the payload was all in text form.

3.4 Additional features of QUIC

3.4.1 Loss Detection and Recovery

For loss detection, QUIC uses both an ACK based method as well as a time-out based method [3]. For the second method an estimated network round-trip time is calculated. The time-out based method is used at the end of a data transfer where Acks won't continuously be sent anymore. A well estimated RTT is important to not retransmit data when the original is still on its way, but also not to delay the retransmission too much for when data is lost.

ACK-based Detection The method of detection uses the different forms of loss detection TCP has. The first is the fast retransmit. In this situation a packet is lost when an ACK is received of a packet that has been sent after the unacknowledged packet. Let us say that a packet with packet number 1 is lost. The client will continue to Ack subsequent packets. When the server gets an Ack of PN 4, fast retransmit is triggered because PN 1 hasn't been Acked yet even after 3 subsequent packets. That threshold for when to trigger fast retransmit can be chosen. This is to prevent retransmits when packet reordering happens during transport.

The second way of detection is the early retransmit. A problem that can occur with the first method, is when the data transmission is almost over. At that point no additional packets will be sent and there cannot be detected if a packet is lost, since the difference in sequence numbers will not exceed the threshold. To solve that an alarm will be set, which will trigger a retransmit when no ACK is received before the timer expires. The calculated time for the alarm will be based on the estimated smoothed RTT or the latest RTT if that one is longer. It also will be resilient towards reordering.

Timer-based Detection First there is the Tail Loss Probe, the way this works is comparable with the early retransmit. Like the name suggests this will be used at end of data transmission where it is harder to detect loss, especially with ACK-based methods. But, this method is not used to retransmit data. It is used to trigger an acknowledgement from the server, as well that the unacknowledged data is not marked as lost yet. When sending a TLP, new data can be included to not interrupt the transmission.

There is also the Retransmission Timeout which will be triggered when the TLP fails to get a response. The RTO packet is considered to be a stronger signal than the TLP but not strong enough to indicate packet loss.

A special case is loss detection on the handshake, since the handshake is needed for the whole QUIC connection. A timer is also used in this situation. When it expires a probe will not be sent to see if the data arrived, immediately all unacknowledged will be retransmitted by the sender to reduce the delay for setting up a connection.

3.4.2 Error Handling

Errors on a connection will always occur, so QUIC allows errors being signalled to the other endpoint. There are different ways to handle errors, one is connection level errors. These can affect a whole connection and can be because of violation of protocol semantics or corruption of state. With a *CONNECTION_ CLOSE* frame these kinds of errors can be reported.

There can also be errors on the stream level, which only affects 1 stream of the connection. This can be handled with the *RST_ STREAM* frame. QUIC also allows application to define their own errors and error codes. These can be carried by the QUIC protocol in the *RST_ STREAM* and *APPLICATION_ CLOSE* frames.

4 QUIC visualisation tool

Currently QUIC is being standardized by the QUIC working group. There are already some version independent features present in the RFC documents. However, there are still things that are loosely defined or undefined. As well as that, there are also implementation choices left open to the developers. A lot of uncertainties are still present and need to be refined.

With a lot of the QUIC specifications still open for discussion, changes can still be made which would improve the protocol. However, just designing the protocol using RFC documents is not optimal. The protocol needs to be tested in practice and also focus on non-routine scenarios where usually difficulties arise.

For the developers there is a substantial amount of items that are left open to interpretation. One of the biggest examples is congestion control, where the protocol allows a congestion control algorithm to be plugged in. Of course the developers would want to use the best congestion control algorithm, but which one is the best? Is there even such a thing as the 'best' algorithm? What if two or more congestion control algorithms perform well in each their own regard, should we switch them out to get better performance?

Currently with HTTP/2 being standardized and it being in use, how well does it perform with QUIC? QUIC provides several primitives for HTTP/2 to use. There is the ability to prioritize streams, QUIC provides a way to set a priority to streams. However HTTP/2 will have to decide which streams to prioritize, in general the application has to define its own prioritization scheme. There are also different kinds of stream types: unidirectional and bidirectional. Bidirectional streams are used for the requests and response, unidirectional can be used as control stream where 2 of them are used. The question is, do these work as intended? Or, are there any limitations which is holding back performance? Compared to using TCP, does QUIC outperform TCP or are there any areas where TCP reacts better than QUIC?

At this point more study is needed into QUIC itself and other dependencies with other protocols. This definitely means the protocol needs to be tested in practice. The QUIC protocol is already very complex and might become even more complex in the future. It will deal with stream/connection level flow control, in-order delivery separate for each stream, integrating TLS for securing the connection, ... When developing a QUIC implementations problems will occur, which would need to get fixed. For this a tool is essential in improving that process. With the QUIC protocol constantly changing there aren't many good tools available, the ones that are available are mostly text based. Using visualizations in this scenario would significantly improve dealing with problems in QUIC. Visualisations can make use of symbols and metaphors which humans process much faster than pure text. Visualisations also present information in a condensed format, pure text can makes us suffer from information overload.

That is why we are developing our own tool that focuses on showing visualisations of data. The primary goal is reduce the time needed to do debugging or discovering differences. But, let us first look at some existing tools.

4.1 Existing tools

4.1.1 Wireshark

There are not many tools available for analysing the QUIC protocol. Most likely because of the reason that the specification can rapidly change, which requires adjustments in the tools. One tool that is available which supports the QUIC protocol, is Wireshark [4].

Wireshark is a tool which can be used for packet capturing and analysis. Low-level information is available of the packets. For some protocols like TCP there are visualizations available, but not for all protocols. QUIC support is only available in the development build of wireshark, it will take some time before it is available in the stable release. Wireshark allows for decryption of the QUIC packets, which is key to analyse the protocol.

The data from a capture of a QUIC connection is pure text-based. There are difficulties with using wireshark to analyse the protocol. For example, tracking when data is being sent over streams when it happens concurrently in wireshark is confusing. Each packet has to be checked manually to which stream it belongs as well as the time stamp.

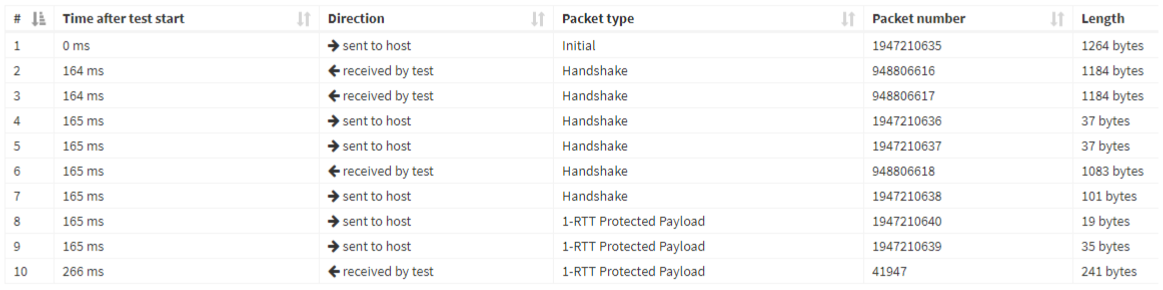
Even if wireshark is showing a condensed version of the packets as an overview of the connection, it can only fit so much data on the screen. If there are any unusual interactions in the connections, it may take some time to find it. It could also happen the user glances over the data and easily miss something that is interesting.

4.2 QUIC Tracker

Another tool that is available is QUIC Tracker [5]. This tool has 2 different goals. The first one is to collect data on popular domains to see if they support QUIC, this is done by checking the `Alt-svc` field in the HTTP header. The second is one is to run test scenarios against the different QUIC implementations that are currently being developed. It also checks if the implementations behave according to the IETF specification.

Compared to wireshark, QUIC Tracker displays the captured information in a more structured way (Figure 10). The table format used in Figure 10 makes it easier to spot who sent what packet. Though the rest of the captured data is displayed in the same way as in Wireshark. For example, looking at the frames that are contained in each packet requires the user to individually select each packet.

[4] https://www.wireshark.org/
[5] https://quic-tracker.info.ucl.ac.be/about



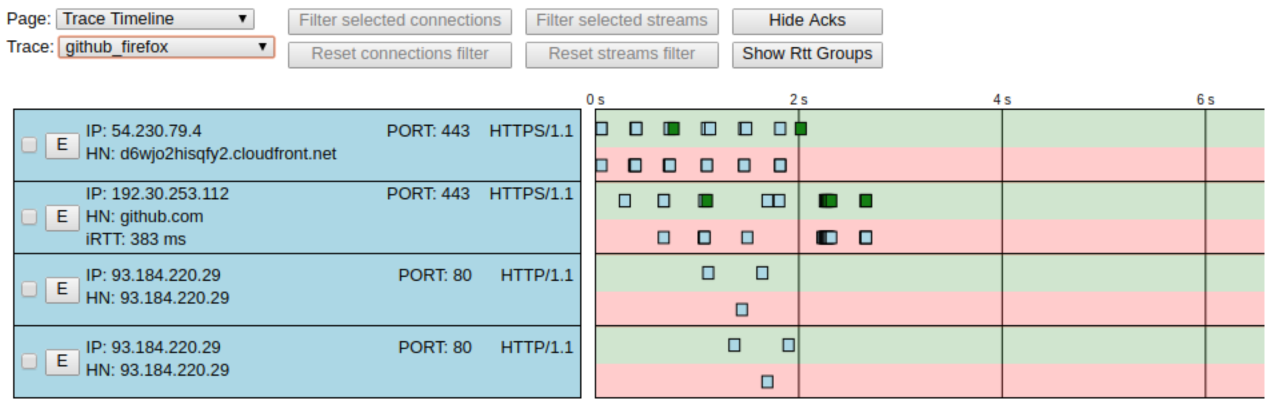
#	Time after test start	Direction	Packet type	Packet number	Length
1	0 ms	→ sent to host	Initial	1947210635	1264 bytes
2	164 ms	← received by test	Handshake	948806616	1184 bytes
3	164 ms	← received by test	Handshake	948806617	1184 bytes
4	165 ms	→ sent to host	Handshake	1947210636	37 bytes
5	165 ms	→ sent to host	Handshake	1947210637	37 bytes
6	165 ms	← received by test	Handshake	948806618	1083 bytes
7	165 ms	→ sent to host	Handshake	1947210638	101 bytes
8	165 ms	→ sent to host	1-RTT Protected Payload	1947210640	19 bytes
9	165 ms	→ sent to host	1-RTT Protected Payload	1947210639	35 bytes
10	266 ms	← received by test	1-RTT Protected Payload	41947	241 bytes

Figure 10: QUIC Tracker

However, as explained earlier the tool captures data on its own by running different test scenarios. It uses its own client which in turn also records the data, it even converts it to a pcap file which can be download. This makes testing for any potential issues easier. The downside is that the tests need to be run separately and need to be completely finished before the data is available. The tool also does not make it easy to compare tests across different implementations.

4.2.1 H2Vis

The following tool, H2Vis [6], is an analysing tool for HTTP/2. This tool sadly doesn't have support for the QUIC protocol but does serve as an example of an tool that focusses on visualising data.



E IP: 54.230.79.4 HN: d6wjo2hisqfy2.cloudfront.net PORT: 443 HTTPS/1.1

E IP: 192.30.253.112 HN: github.com iRTT: 383 ms PORT: 443 HTTPS/1.1

E IP: 93.184.220.29 HN: 93.184.220.29 PORT: 80 HTTP/1.1

E IP: 93.184.220.29 HN: 93.184.220.29 PORT: 80 HTTP/1.1

Figure 11: H2Vis

The way that data is being displayed here, is what were trying to achieve with this thesis for QUIC. At first glance we can immediately see how many packets each endpoint has sent. Looking at the colour of packets being displayed, we can quickly see what kind of packets they are. It is also possible to display the individual streams, which groups the

[6] https://github.com/DaanDeMeyer/h2vis

data per stream. Instantly showing how each stream was utilized. There are also filters available for showcasing/hiding specific types of information, making it easier to spot abnormalities.

This type of tool for QUIC would make debugging a lot easier. Before we showcase the tool we made, let us look first at the ideal features of an analysing tool for QUIC.

4.3 Requirements for ideal tool

An ideal tool for analysing the QUIC protocol during its development stage but also for when its deployed needs a healthy amount of features. Having very detailed information of the protocol like in wireshark is still needed, however it also should have ways to visualize that data. In this section we will look at what features would be present in an ideal tool [8].

R1 Handling different kinds of inputs:
First there is tcpdump output in the form of pcap files. Since this is a separate tool it can be used in combination with any of the QUIC implementations. Though with QUIC traffic being heavily encrypted this may have issues. Wireshark does allow data to be decrypted, but only if the SSL keys can be gathered. Most QUIC implementations have their own log output, which can also be used as input. Log files may be preferred since they can provide additional information: decisions that are made in connection with congestion control, flow control, ..., that are not present in pcap files.

R2 Compare multiple results
During the development of QUIC there are a lot of different implementations which are built in different ways. In this stage of development the focus is on making sure the different implementations can connect with each other. Issues arise all the time so finding a solution is important. Having the ability to compare different results can significantly speed up the process. When comparing a successful connection with a faulty one, differences can be spotted easily. For later when experiments may be done on different congestion control algorithms, comparisons need to be done.

R3 Provide chronological view of a connection:
As mentioned earlier, one of the important things is 2 endpoints being able to communicate with each other. So analysing how the communication is structured is going to be important. There needs to be a high level view of it in form of packets, which can point out any issues on connection level like incorrect header information. Since QUIC also supports use of streams to be able to sent data concurrently, the individual streams need to also be present.

R4 Provide detailed look at a single stream, how it operates
An important stream is stream 0, since it is the control stream where communication happens for configuring or managing the connection. A stream should be able to be viewed in detail as well. For example, when doing research in flow control. Stream 0 will be used to send flow control frames which will contain important information as in how many streams can be opened. It can also showcase where flow control is stepping in to delay a data transfer.

R5 Provide detailed look at a single QUIC packet
Knowing what kind of data is present in a packet is equally important. It will help with providing more information on the interaction between endpoints. Looking at a packet that the client sent and comparing that to the response received from the server could help with detecting violations against the QUIC Protocol.

R6 Highlight unusual things/error detection
The tool can go even a step further and automatically highlight errors. There are QUIC frames which are used for sending all kinds of errors. After loading a file in the tool, those kind of errors can be highlighted. A researcher will immediately see where the problem is in the connection. Another situation is when congestion control is active. The tool can show where retransmits are happening. At first glance the researcher can see how regular retransmits are and conclude if there is some kind of underlying issue.

R7 Provide detailed look at interaction between HTTP/2 and QUIC
One of the protocols that QUIC will be used for is HTTP, more specifically HTTP/2. QUIC already provides some features which HTTP/2 can make great use of: stream multiplexing, stream-level flow control, stream prioritization, ... A researcher may want to know if these features are being used as intended or if there are any issues like a stream being blocked when it has a high priority.

R8 Capture data within the tool, instant testing
This tool is primarily for studying the QUIC protocol and comparing different implementations and scenarios. But capturing data from multiple programs or using server logs which are structured differently for each implementation, is not easy to maintain. A nice to have feature would be the ability to capture data from within the tool and automatically visualize the data when it is received. A researcher can then in real-time follow the interaction between client and server. At the same time, if there are any anomalies in the connection, they can be instantly spotted without having to wait for the connection to finished.

R9 Sharing of data
Capturing and displaying data is one thing, data is often shared between researchers as well. This is for comparing their findings or to confirm it is correct. Graphs and all source data should be able to be shared, which can be included in their research paper to support their claims. Allowing the tool itself be shared and the state it is

in can help with discovering anomalies, perhaps even allowing multiple people to change settings at the same time.

4.4 QUIC-Vis

QUIC-Vis is the website that has been made for this thesis. In this section we will discuss the different components of it and the features.

4.4.1 Data sources

In order to create visualizations we will need to insert data first. For this we use two different input files (**R1**).

Wireshark traces The first one will be in the form of wireshark traces. The file format used is `pcap`, but in order to parse the data a few more steps have to be executed after capturing. At first, we planned to directly use pcap-file with help of a library. However, after searching for some time there was no such library found. It is also possible to create our own library to support pcap files, though it would have taken quite a lot of time to implement. So we went with a different approach. To read the file, we can use Tshark to convert it to a `JSON` file. Tshark is the command line version of wireshark which supports the same functionality as wireshark.

As mentioned earlier, QUIC encrypts all of the payload and parts of the header after the handshake. The captured data will also be encrypted. So the data needs to be decrypted first, it is possible to do this within wireshark. In order to decrypt traces, the TLS session keys need to be retrieved of the connections. Most TLS implementation support the exporting of keys, some server implementations use the environment variable `SSLKEYLOGFILE` and others allow a paremeter to be provided when running the server. We use Tshark to first decrypt the files by linking the TLS keys in the form of a `keys` file. When it is finished we have a decrypted network trace of QUIC as a JSON file, this we can use as input for the website.

Server logs Wireshark traces provide a lot of information about QUIC, but it's only able to do that about the protocol itself, the packets and frames that are being sent. Behind the scenes happens a lot more, like deciding when packet is lost, current state of flow control, stream prioritization settings, ... That kind of information is stored within servers and clients and is present in server/client generated log files.

Using these files as input for the website would be more interesting than wireshark traces because of the extra information. However, each client/server has its own log structure. We parse these log files as text files which takes time implement, to correctly gather all the data. In order to support all the QUIC implementations we would need to spend a lot of time on that. Given the time constraint we decided at first to support 3 implementations, the choices made are explained in the following section. Due to time constraints we only managed to support Ngtcp2 and Quicker. We are allowing support for more QUIC implementations to be added later in the future.

4.4.2 Website

The visualizations will be done with a website. It will read the data from the files, process it and create different visualizations. With it being a website, we make use of HTML, CSS and SVG. HTML will be used to represent the textual content whilst SVG is used for the visual representation of the data, CSS is for styling both of those elements.

For the basis of the website we use the VueJs[7] library. VueJs is a JavaScript framework to design user interfaces which are reactive. Being reactive means that any changes to the data or settings while the website is loaded, the page will automatically be updated and display the changes. Only the parts/components of the website which are changed will be refreshed, this keeps it efficient since the whole page doesn't need to be reloaded.

For managing these changes and keep an overview of the current state of the application, we make use of Vuex [8]. Vuex is a state management library officially supported by the Vue team. This allows the state of the application to be manipulated in a centralized place, making it easy to let the different components interact with each other.

Since we are going create complex visualizations with SVG, we want to spend as little time as possible on creating the simple parts. So we are going to use the D3 library [9] which provides a lot of elements like scales, axes and interactivity. With this library we can create our own complex graphs and charts.

Rather than implementing it all in JavaScript, we make use of Typescript [10]. Typescript is a superset of JavaScript which is compiled to JavaScript in order to run. As the name suggests, Typescript supports typing. With it needing to be compiled first, a lot of errors can be found and corrected during that step. This makes the development process significantly easier since most errors are easily spotted.

4.4.3 Timeline

The timeline page is the main visualization of the website. It showcases the different QUIC connections from the input files and gives high- and low-level information about each connection (**R2**). This page consists of a couple different components which we will discuss in detail. At first we will discuss the mockups we made, after that we will describe any changes that were during development.

[7] https://vuejs.org/
[8] https://vuex.vuejs.org/
[9] https://d3js.org/
[10] https://www.typescriptlang.org/

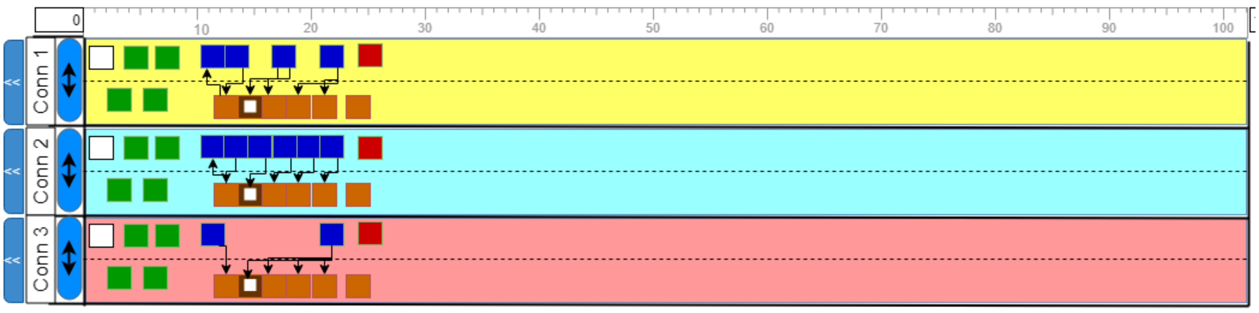


Figure 12: Mockup: Timeline of connections

Timeline Chart The main components of the page is the timeline chart (**R3**). For each connection it will display the different packets sent over time by default. Packets will also be grouped based on whom the sender is, client or server indicated by green and red.

The axis at the top display the time domain in ms, which can be adjusted by the two input fields at each end. It is also possible zoom with scrolling, after which we can translate the view as well.

We can also look at a connection in more detail by displaying the streams that are used in the connection. For each packet, the frames that are contained within will be show in the corresponding stream. When multiple frames of the same packet are sent in the same stream, those frames will be displayed underneath each other.

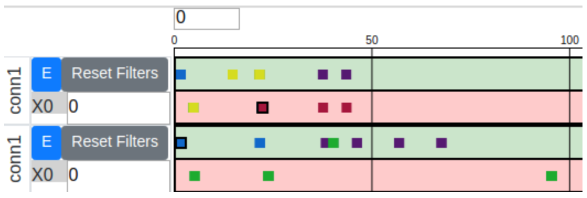


Figure 13: Timeline of connections with send/receive lane

During development, we decided not to give each connection a different background colour in the timeline. We changed it to giving each connection a send and receive lane coloured green and red respectively (Figure 13). This makes it easier for following a connection, seeing which packets were sent and which ones were received. We also decided to not include the arrows drawn between a packet and the ack of the same packet. This could clutter the timeline making it hard to use.

On the left of the chart we have the identifiers for the input files and the QUIC connections that are available. On there we can hide/show the streams in the chart (**R4**), when the streams are shown we can also individually filter them out. It is also possible to reset the stream filters. It might also be interesting to compare to input files of the same

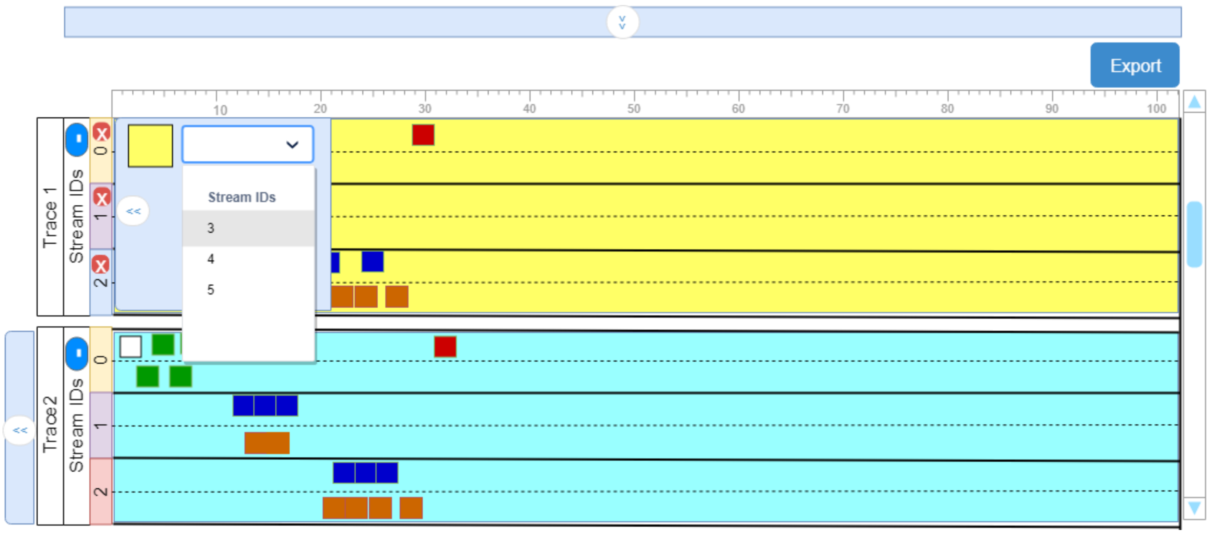


Figure 14: Mockup: Timeline of streams within a connection

connection, however those are not always synchronized. Because of that, there is also an input field to manually set the relative start time.

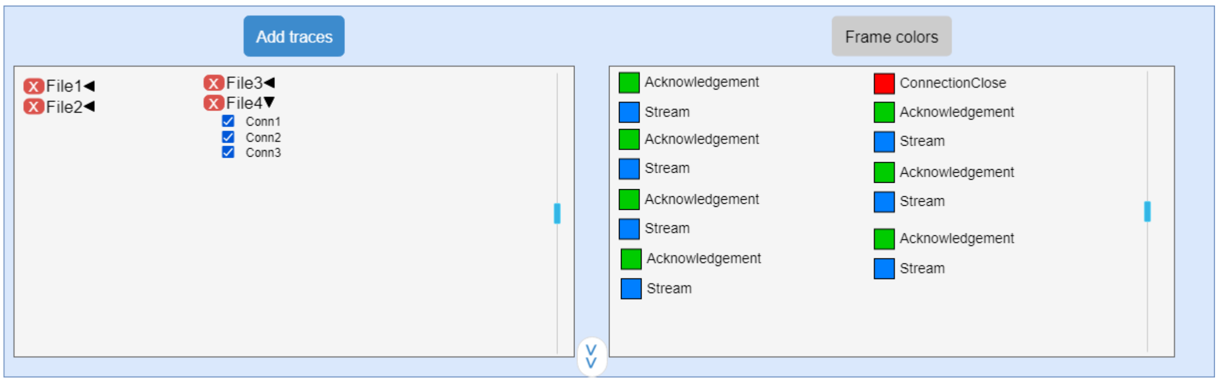


Figure 15: Mockup: General Settings

General settings First when clicking on viewing the settings, we can see the list of files that are loaded. In there we can filter out any of the connections in there. Next to the list of files we see the different frames types and the colours that are used to represent those in the timeline chart below.

Next to viewing settings there a couple buttons which can be used to select a colour scheme. The default scheme sets a colour for every frame type. The flow control scheme only sets a colour for frames that are used for flow control. The errors scheme only sets a colour for error frames (**R6**). With using these colour schemes, problems/anomalies can easily be spotted.

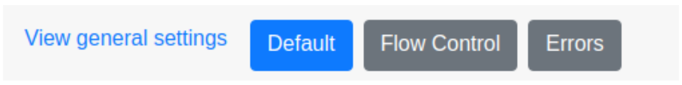
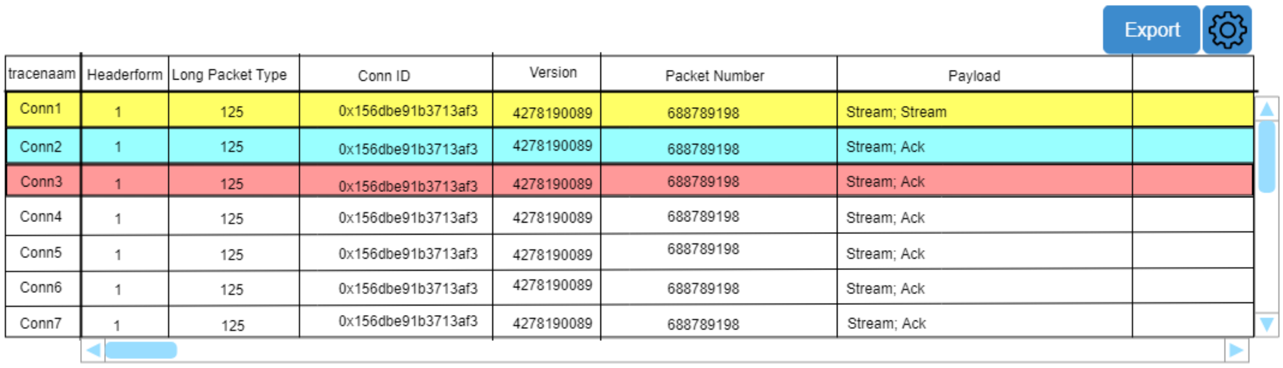


Figure 16: Buttons to filter specific frames



tracenaam	Headerform	Long Packet Type	Conn ID	Version	Packet Number	Payload	
Conn1	1	125	0x156dbe91b3713af3	4278190089	688789198	Stream; Stream	
Conn2	1	125	0x156dbe91b3713af3	4278190089	688789198	Stream; Ack	
Conn3	1	125	0x156dbe91b3713af3	4278190089	688789198	Stream; Ack	
Conn4	1	125	0x156dbe91b3713af3	4278190089	688789198	Stream; Ack	
Conn5	1	125	0x156dbe91b3713af3	4278190089	688789198	Stream; Ack	
Conn6	1	125	0x156dbe91b3713af3	4278190089	688789198	Stream; Ack	
Conn7	1	125	0x156dbe91b3713af3	4278190089	688789198	Stream; Ack	

Figure 17: Mockup: table of packet contents

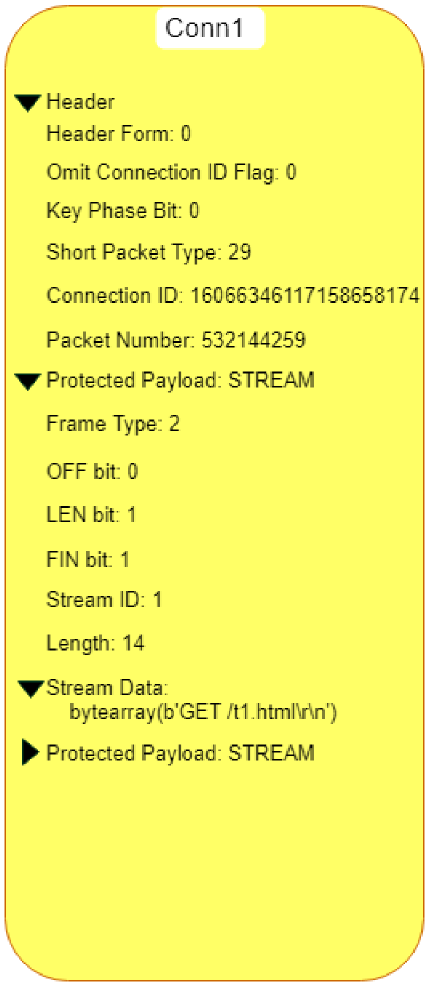
Table Underneath the timeline diagram there is a table which contains header information of a packet for each connection. Other packets can be selected by clicking on them in the timeline chart, selecting a packet will only change the selected packet of the same connections. In this table we can compare different header information, flags that are used, type of packet used ... .

We can also change the table by filtering the different columns. This makes it easier to look for something specific. It is also possible to sort the different columns for comparing differences.

Detailed Packet info Right of the timeline chart we have can see detailed info of a packet (**R5**). This will contain a lot of information about the packet like sender & receiver, packet number, frame data, ... If the selected packet is comes from a server log file where there is extra information, that info will also be displayed. Selecting a packet is done the same way as with the table, only the last clicked packet will be shown.

4.4.4 Sequence

The second page shows a sequence diagram of a connection. There needs to be at least 1 file and connection selected before it displays the diagram. If one file is selected the RTT will be simulated. It is possible to manually set the RTT which will be used to draw the sequence diagram. A second file can also be selected to get a more accurate sequence diagram, of course both files need to contain information about the same connection. There are also filters present to hide/show certain information. The user can also click


Conn1

▼ Header

Header Form: 0

Omit Connection ID Flag: 0

Key Phase Bit: 0

Short Packet Type: 29

Connection ID: 16066346117158658174

Packet Number: 532144259

▼ Protected Payload: STREAM

Frame Type: 2

OFF bit: 0

LEN bit: 1

FIN bit: 1

Stream ID: 1

Length: 14

▼ Stream Data: bytearray(b'GET /t1.html\r\n')

▶ Protected Payload: STREAM

Figure 18: Mockup: Detailed information about packet contents

on the arrow or text to get a clear view of that packet, with double click the user can revert the diagram to its original state.

During development, we encountered an issue for keeping the timestamps in the correct order without cluttering the screen with a lot of arrows. We ended up with using a 2 pass algorithm where in the first pass we get the packets of each file separately and sorted according to send/receive time. We then go through each list of packets and determine the y-coordinate where the arrow would start/end. If 2 coordinates are too close, we add a margin to keep them separated. Once that is finished we start with the second pass, where we draw for each packet the arrow with the coordinates determined in the first pass.

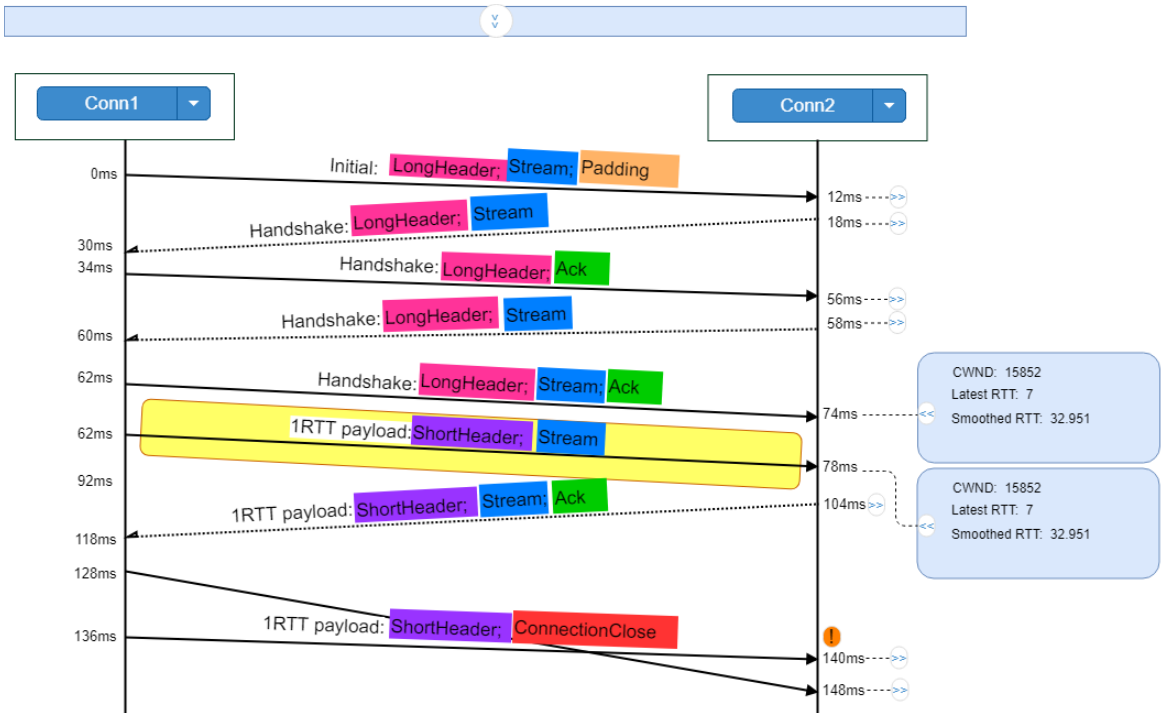

Conn1
Conn2
0ms
30ms
34ms
60ms
62ms
62ms
92ms
118ms
128ms
136ms
Initial: LongHeader; Stream; Padding
Handshake: LongHeader; Stream
Handshake: LongHeader; Ack
Handshake: LongHeader; Stream
Handshake: LongHeader; Stream; Ack
1RTT payload: ShortHeader; Stream
1RTT payload: ShortHeader; Stream; Ack
1RTT payload: ShortHeader; ConnectionClose
12ms
18ms
56ms
58ms
74ms
78ms
104ms
140ms
148ms
CWND: 15852
Latest RTT: 7
Smoothed RTT: 32.951
CWND: 15852
Latest RTT: 7
Smoothed RTT: 32.951

Figure 19: Mockup: Sequence diagram for 2 traces

4.5 Not implemented requirements

Due to the limited time available we were not able to implement all the requirements in Section 4.3. We give an overview of those requirements that are not implemented and why:

R7 Provide detailed look at interaction between HTTP/2 and QUIC:
HTTP/2 itself is a very complex protocol, providing support for HTTP/2 would require a lot of time. Also integrating HTTP/2 support in QUIC is not a priority at this moment. The main focus is making clients and servers being able to communicate with each other and improving the QUIC connection itself.

R8 Capture data within the tool, instant testing
A lot of work went into supporting pcap and log files, which did not leave much room to support a live capture and rendering. The challenge is to be able link a file to QUIC-Vis where data of a connection will be available and then rendering the data as it comes in.

R9 Sharing of data:
Sharing of data can be partially done because QUIC-Vis is a static website. The website and the files can be hosted on a web server and researchers will be able to access the same files with the url. However, any settings that are customized will

be linked to user and cannot be shared to other users. This was not implemented simply due to the lack of time.

5 Results

In this section we will describe how QUIC-vis can be used to do research on QUIC. A couple different scenarios are chosen in which we use QUIC-vis to analyse and/or discover problems within a set of QUIC implementations. We will showcase how QUIC-vis as a tool can be used and what the benefits are of using it over other tools that are currently available.

For performing tests we will use three different QUIC implementations: Ngtcp2[11], Quicker[12] and Quant[13]. There are more QUIC implementations available, but due to the limited time available to support server logs, not every implementation supporting draft-11, not being to build all implementations locally, ... ,we chose to use these three. These QUIC implementations will be run locally where we will use different methods to simulate network-like behaviour.

For gathering test data we use two different methods. The first one is capturing a trace with Wireshark. In order to fully utilize a Wireshark capture, we will need to export the TLS-keys used in the connection. In order to use the file in the tool, we will first have to convert it to a JSON file. This allows QUIC-vis to support all the QUIC implementations which support TLS key export. This is still tedious to do and doesn't provide much of a benefit over Wireshark.

The second method however does, where QUIC implementations provide their own output of the connection. With enough information available like header info, payload info, timestamps, ..., this output can be used to reconstruct a network trace as if it were captured with Wireshark. Not only does it prevent it the need to set up wireshark for capturing data and the collecting of TLS keys, these QUIC clients/servers will have more information available then just about the QUIC packets. State of flow control, labelling which packets were lost and retransmitted are a few pieces of information that is not transmitted over the connection. So Wireshark will not have that information available nor can server logs be parsed in Wireshark.

The set-up we use for the servers is a set of docker containers we have build from `https://github.com/moonfalir/quic-docker-builds`. For the server we build docker containers from the following directories: `quicker`, `ngtcp2` and `quant`. After building the containers, they can be started with the `start-container.sh` script in the corresponding directory. When running these docker containers, we have to manually start the servers.

For the clients used to run different scenarios, we build the container from the `quicker-clients` directory. When the container is build we can also make use of the `start-container.sh` script start the container. When running the script, the container will consist of a basic Quicker client. The `start-container.sh` script also accepts the following parameters:

[11] https://github.com/ngtcp2/ngtcp2
[12] https://github.com/rmarx/quicker
[13] https://github.com/NTAP/quant

`-d`: send duplicate packets, `-i`: send 2 initial packets, `-f`: limit flow control, `-o`: perform 0-RTT handshake.

5.1 Handshake

QUIC-vis has a sequence diagram page where we can take a closer look at the interaction between two endpoints. With this we will look at two scenarios of performing a handshake in QUIC.

5.1.1 incorrect 0-RTT parameters

QUIC allows clients to perform a 0-RTT handshake to servers which it already has connected to before, allowing the client to immediately send data. This is possible due to clients and servers saving the transport parameters which are advertised during connection set-up. In this scenario we will look at what happens when those parameters are incorrect.

In order to perform this test we will first have to establish a connection to the server with the 1-RTT handshake, during which we will save the parameters in a session file. When we have the session file we manually change the data in it, so we have incorrect transport parameters. Then we will establish another connection to the same server using this session file, this is the connection that will be captured.

In the QUIC RFC document, it specifies that a 0-RTT handshake can succeed but 0-RTT data can not be sent if some transport parameters have a value of 0. So we expect so see a 0-RTT handshake to be executed and see no 0-RTT protected packet being sent at the beginning.

What we first changed in the transport parameters in order to send incorrect parameters, is changing the max_data value to 0. However, with changing this value, the Quicker client does not start a connection. This is because the client itself would not be able to send 0-RTT data even if a connection is made. When trying to send the initial packet, Quicker first checks if the server's flow control allows it. It sees that max data is 0 so it creates a blocked frame. However, flow control frames are not allowed in handshake packets, so it doesn't send any packets. Looking at the QUIC specification, handshake frames are exempt flow control. So the Quicker client is allowed to perform the handshake.

We also tested with changing the initial max_stream_bidi_id. In Figure 20 we can see the result of the test with Quicker. In here we do see the client perform a handshake, we also see that there is no 0-RTT protected packet being sent. This is because the http request would be sent on a different stream, but the client is not allowed to open another stream. The new stream would be a bi-directional stream and the limit of that is 0. When the handshake is completed, the server responds with a new set of transport

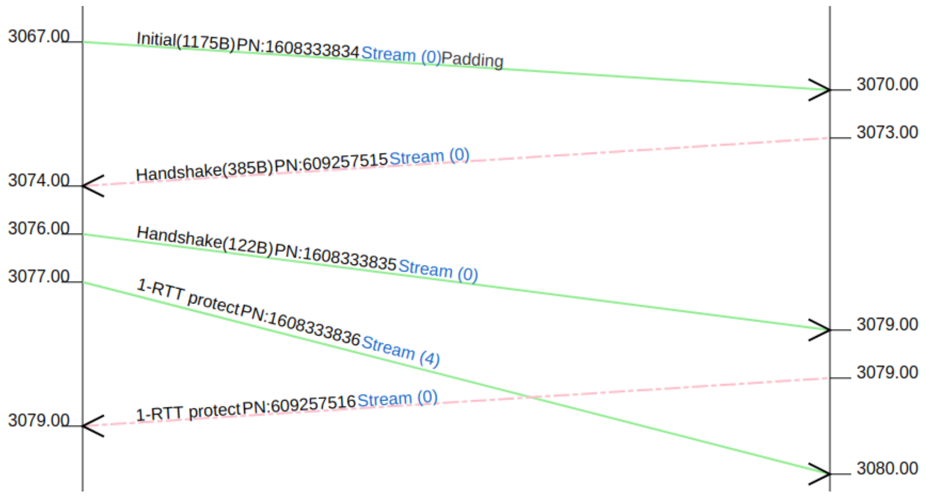

3067.00
Initial(1175B)PN:1608333834Stream (0)Padding
3070.00
3073.00
Handshake(385B)PN:609257515Stream (0)
3074.00
3076.00
Handshake(122B)PN:1608333835Stream (0)
3077.00
1-RTT protectPN:1608333836Stream (4)
3079.00
3079.00
1-RTT protectPN:609257516Stream (0)
3079.00
3080.00

Figure 20: Sequence of incorrect 0-RTT parameters

parameters. This allows the client to send the http request within a 1-RTT protected packet.

5.1.2 Duplicate initial packet

For a QUIC connection to be set-up, the client and server need to establish TLS-keys to encrypt data. In order to do this and open a connection, the client sends a packet of type `Initial`. In this scenario we will experiment when the client sends two initial packets at the same time.

To perform this test we will first have to make modifications in the QUIC client in order to ensure that this behaviour will happen. Basically when opening a connection, we will have to copy the initial packet and send the copy as well to the server. However, we need to make sure the packet number is different since duplicate packet numbers in a QUIC connection is not allowed, we will look at that scenario later in Section 5.3.

We expect that the second initial packet will be ignored. The server might still process the packet but will find that both initial packets are equal in content. Since the server already starts with the handshake after the first packet, there is no reason to send additional handshake packets for the second one.

When we look at the sequence diagram of the Quant server (Figure 21) we can see 2 initial packets are sent. What follows is what we expect, the server replies to the first initial packet and ignores the second one. The server does send 2 handshake packets in reply but they both have a different size, this is done because of the required data to perform the handshake larger is than the allowed packet size. If we would have seen 2 handshake packets with the same size, then the server would have replied to both initial

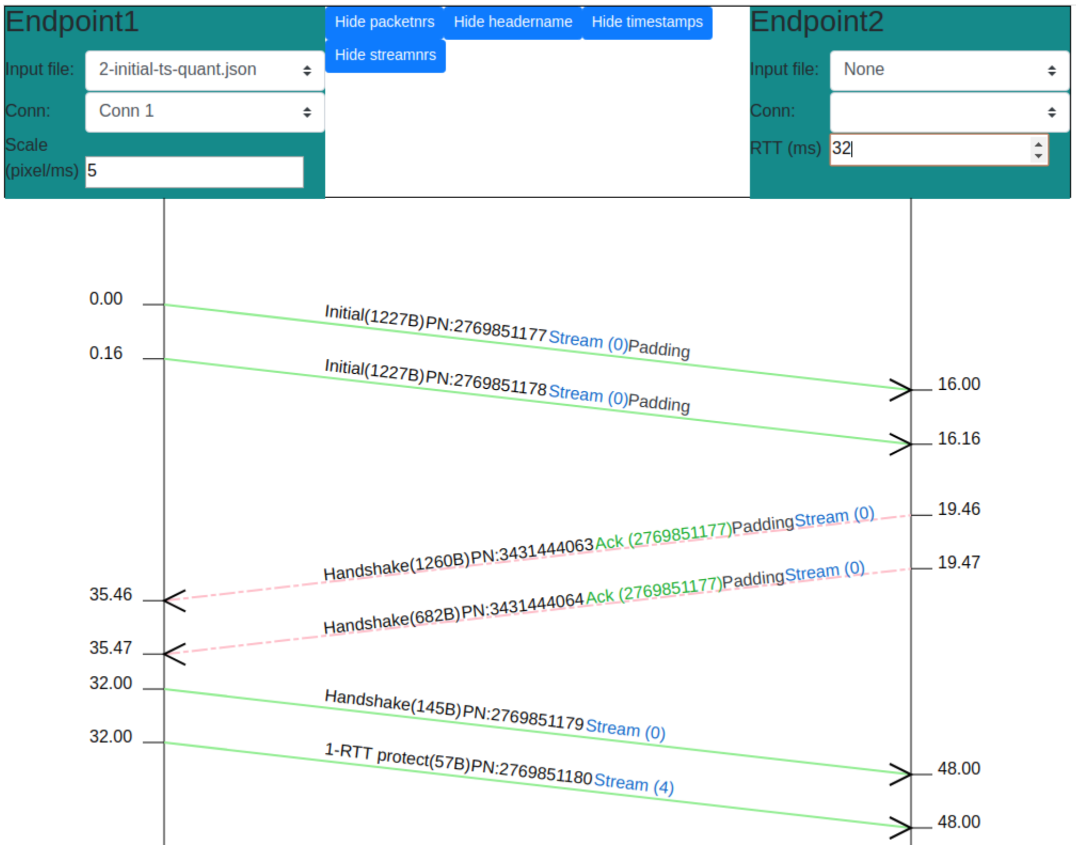

Endpoint1
Hide packetnrs
Hide headername
Hide timestamps
Hide streamnrs
Endpoint2
Input file: 2-initial-ts-quant.json
Conn: Conn 1
Scale (pixel/ms) 5
Input file: None
Conn:
RTT (ms) 32
0.00
Initial(1227B)PN:2769851177 Stream (0)Padding
16.00
0.16
Initial(1227B)PN:2769851178 Stream (0)Padding
16.16
19.46
Handshake(1260B)PN:3431444063 Ack (2769851177)Padding Stream (0)
35.46
19.47
Handshake(682B)PN:3431444064 Ack (2769851177)Padding Stream (0)
35.47
32.00
Handshake(145B)PN:2769851179 Stream (0)
48.00
32.00
1-RTT protect(57B)PN:2769851180 Stream (4)
48.00

Figure 21: Sequence diagram with 2 initial packets

packets. The handshake finishes and the client sends a http request. We see a similar behaviour from the other servers.

5.2 Flow control

QUIC-vis also has a time-line diagram where we can quickly see an overview of the connection. On this page we can also select filters for the colours of packets. One of these filters can be used to highlight flow control frames and see any problems that occur related to flow control. To use this we can look at the following scenario: After the connection is set-up, the client sends both a Max_stream_data frame and a Max_data frame. However, the advertised size in Max_stream_data will be twice the size advertised in the Max_data frame. After this, the client will then request a resource that has a size which lies in between the Max_data size and the Max_stream_data size.

For this, we need to implement this behaviour in the client. Once the handshake has been completed, a packet will be sent containing both a Max_stream_data and Max_data frame. After that packet has been sent, the request of the resource will be sent.

What we expect to happen in this scenarios is when the amount of received data reaches the Max_data size, the connection-level flow control will step in and block the server from send any more data. The server will then notify the client with a `Blocked` frame.

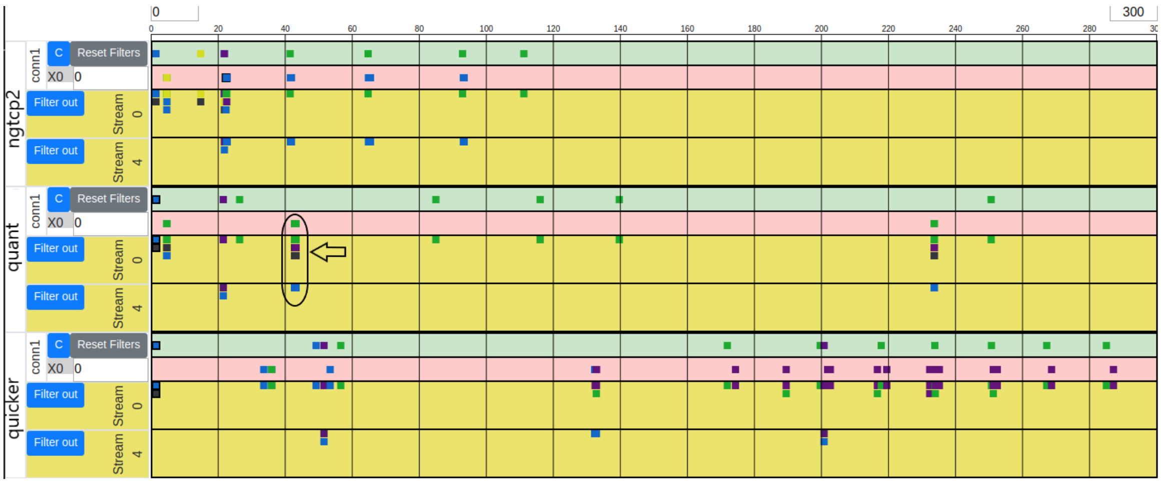


Figure 22: Timeline graph with flow blocked scenario

In Figure 22 we see the captures of this scenario. For this we will look specifically at flow control frames (coloured in purple) and as we expected we do see quite a few flow control frames towards the end, for two of the servers (Quicker and Quant). When selecting such a packet we see that it is a blocked frame, signalling that the connection-level flow is blocked. A difference between Quant and Quicker, is that Quant still includes a stream frame when sending a blocked frame. The circled packet in the diagram contains multiple frames, which we can see in the stream-level part of the diagram. In that packet there is an acknowledgement (green), a blocked frame (purple), a padding frame (black) and stream frame (blue) for stream 4. This should not be allowed since the server then goes over the maximum advertised data.

The Ngtcp2 server does nothing when the flow is blocked. It just stops sending data, it doesn't send blocked frames to indicate that connection-level flow-control is blocking the server. It is not enforced to send blocked frames when the flow is blocked. However, without those frames it can be difficult to find out what the problem is.

We also see multiple blocked frames are sent with the same offset value, which according to the IETF specification should not be done, Quicker even sends more blocked frames in shorter interval (Figure 23). The client acknowledges these blocked frames, which are the green coloured packets in the top row. The server should know that the client knows that the flow is being blocked on its side. Sending those extra blocked frames causes extra overhead which is not needed.

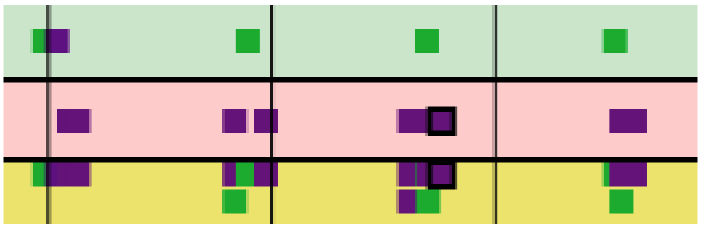


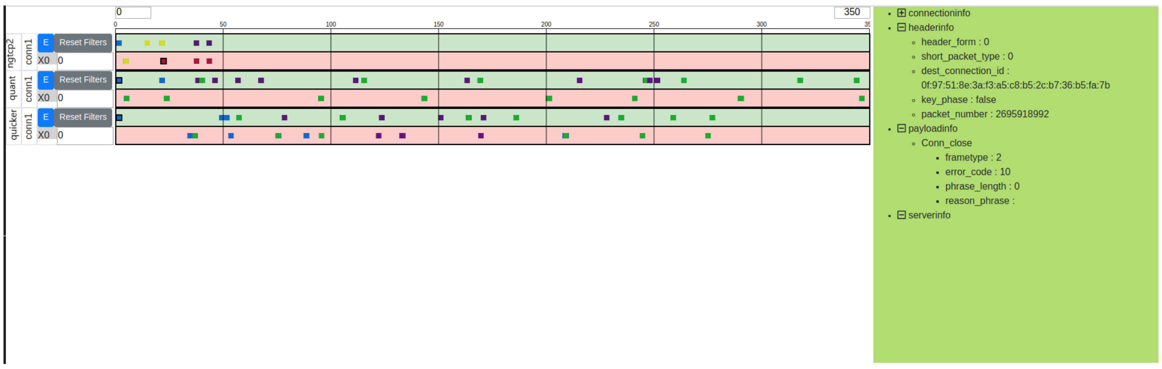
Figure 23: Timeline of quicker

5.3 Duplicate Packet Numbers

Another scenario we can look at is where packets that are being sent, are duplicated. This can happen if the client does not take into consideration that retransmits need to be done with different packet numbers. On-path middleboxes can also decide to send duplicates. So for this scenario we will be duplicating packets when the handshake has been completed and see what kind of impact it has on the connection.

To perform this test we will first have to make modifications in the QUIC client in order to ensure that this behaviour will happen. Basically when the handshake is complete, we will have to copy the short header packets and send the copy as well to the server. We don't not change anything about the copied packet this it to ensure an exact duplicate is sent.

We don't expect much to happen apart from packets being duplicated. Currently the QUIC standard is against reusing packet numbers since it is not clear when acknowledging, if the original packet arrived or the retransmitted one. However, it is not specified that the connection needs to be closed or an error needs to be sent. On the other hand, a duplicate packet can be a signal of a stateless reset. It is only a stateless reset if the last 16 octets of the duplicate packet is equal to the pre-defined stateless reset token.



- ⊞ connectioninfo
- ⊟ headerinfo
  - header_form : 0
  - short_packet_type : 0
  - dest_connection_id : 0f:97:51:8e:3a:f3:a5:c8:b5:2c:b7:36:b5:fa:7b
  - key_phase : false
  - packet_number : 2695918992
- ⊟ payloadinfo
  - Conn_close
    - frametype : 2
    - error_code : 10
    - phrase_length : 0
    - reason_phrase :
- ⊟ serverinfo

Figure 24: Timeline graph with duplicate packets scenario

If we look at Figure 24 we can see 2 of the 3 connections behave as expected. These are the Quicker and Quant servers. Both servers will just ignore the duplicates and only

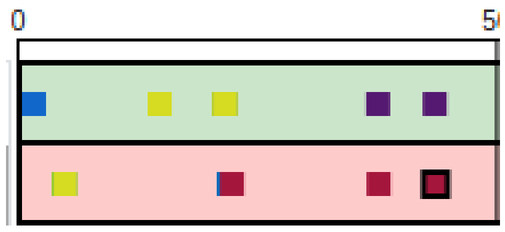


Figure 25: Timeline of ngtcp2

respond on the first one received.

However Ngtcp2 behaves differently, in the timeline (Figure 25) we see a couple of red packets which indicate some sort of error frame is in the packet. When we select one of these packets, we see that a connection close frame is sent. This connection close frame has the error code 2, which stands for protocol violation. So Ngtcp2 will stop the data transfer as soon as a duplicate packet arrives. The RFC specifies that packet numbers must not be reused, but it doesn't say a connection should be closed upon receiving duplicate packets. What is also interesting, is that there are multiple packets with a connection close frame and upon further inspection they all have the same packet number.

5.4 Packet reordering

An interesting situation to look at is when the connection experiences an inconsistent latency, aka jitter, where the order of packets arriving is not the same as the order of packets being sent. Packet reordering can be a problem on the internet, which can have an impact on performance. This is the case with TCP where TCP makes sure all packets are delivered in the correct order to the application. With QUIC the situation is improved due to the use of streams, where the order of packets is not regulated but the order of stream data is.

In order to emulate inconsistent latency we can make use of TC where we can give a random delay to each packet. We will look at a bad case scenario where the delay value is chosen randomly, this can be done with this command: `tc qdisc add dev lo root netem delay 10ms 2ms`. We chose these values to analyse a common situation, where the network is relatively stable with some variance in latency. With the delay being applied twice to a packet, a packet can be delayed by 18ms and up to 22ms. Once that is set-up we can open a connection and observe how the data transfer takes place.

In this situation we may see pre-emptive retransmissions of packets, since at this moment QUIC uses a basic version of loss detection mentioned in Section 3.4.1. Packet reordering is taken in to account in loss detection but not for higher degrees of reordering. For example, packet 1 can arrive after packet 2 and 3 have arrived without triggering a retransmission. But, if packet 2,3,4 and 5 arrive before packet 1, packet 1 would be very likely lost. This could have a high impact on performance where the server will do multiple retransmissions before transmitting new data. Currently this is not the main focus of the QUIC working group, but later it will be an important topic for research.

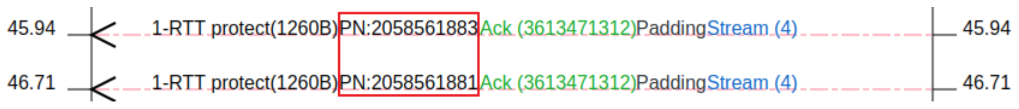

45.94
1-RTT protect(1260B)PN:2058561883Ack (3613471312)PaddingStream (4)
45.94
46.71
1-RTT protect(1260B)PN:2058561881Ack (3613471312)PaddingStream (4)
46.71

Figure 26: Sequence diagram with 1 file

When we look at a wireshark capture of Quant and look at the visualisation, we don't see anything special at first glance. When we look closer at it, we see in Figure 26 that packets are being reordered due to packet numbers not being adjacent. If we would try spot this in Wireshark, we would also have to look at the individual packet numbers. However, we do not know if packets are being reordered by the server or during transfer.

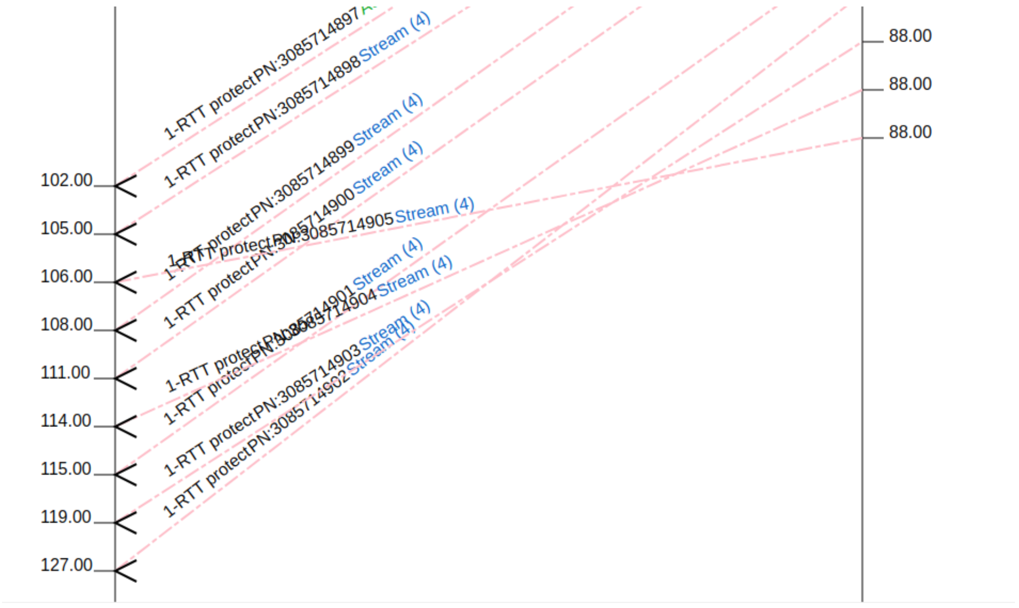
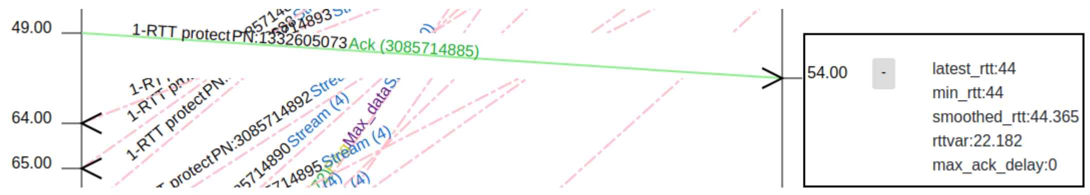

1-RTT protectPN:3085714897
1-RTT protectPN:3085714898Stream (4)
1-RTT protectPN:3085714899Stream (4)
1-RTT protectPN:3085714900Stream (4)
1-RTT protectPN:3085714905Stream (4)
1-RTT protectPN:3085714901Stream (4)
1-RTT protectPN:3085714904Stream (4)
1-RTT protectPN:3085714903Stream (4)
1-RTT protectPN:3085714902Stream (4)
102.00
105.00
106.00
108.00
111.00
114.00
115.00
119.00
127.00
88.00
88.00
88.00

Figure 27: Sequence diagram with 2 files

After loading both the client en server logs from the Ngtcp2 connection we get a very different result. We can immediately see in Figure 27 that packets are being reordered since there are arrows crossing each other. If there wasn't any reordering the arrows would be nice in parallel. It is also the case that the reordering happens during transfer, we can see this at the right side where the last 3 packets are being sent from. The order of the packets that are leaving are in ascending order of packet numbers. But, are there any packets that are considered lost due to the reordering?

With QUIC not reusing packet numbers for retransmissions, detecting retransmissions in wireshark captures does require some extra work. It is possible to do by inspecting packets individually and looking at the offset value of the stream frames. If an offset



latest_rtt:44
min_rtt:44
smoothed_rtt:44.365
rttvar:22.182
max_ack_delay:0

Figure 28: Extra information available in log from Ngtcp2

value appears more than once, we can determine that a retransmission has happened. However, server logs can easily provide information about packets being lost and retransmitted. In Figure 28 we can see some information that the Ngtcp2 server provides. These are calculated RTT values which are used in loss detection. If there were any packets considered lost, we would see the packet numbers of those packet in there as well. Since we don't see any, there were no retransmission. So Ngtcp2 is quite resilient towards low levels of reordering.

5.5 Network interruption

In this scenario we will be looking at when the connection between two endpoints is interrupted. This could be of a node, which is part of the network, rebooting. During that time neither of the endpoints can communicate with each other. It will be interesting to see how these endpoint behave during this situation and how/if they can recover from that situation.

In order to test this we will first limit the throughput to give us ample time to interrupt the transfer. For this we use the following command: `sudo tc qdisc add dev lo root handle 1: htb default 12 && sudo tc class add dev lo parent 1:1 classid 1:12 htb rate 10kbps ceil 10kbps`. This will limit the bandwith to 10 kbps for each endpoint. After this we can instantiate a data transfer between the two endpoints. In the middle of this we can introduce, with the help of the following command: `sudo tc qdisc add dev lo root netem loss 100%`, 100% packet loss. We will then leave it like that for a couple of seconds where after we will stop with dropping packets and continue the transfer.

When the connection is interrupted we will expect retransmits coming from the server, due to a time-out alarm firing. When the connection is restored, we expect a brief period where the connection is trying to recover. During that time some retransmits may happen, however this time they will be received and acknowledged. Depending on how long the connection is interrupted, the endpoints might close the connection.

When we look at this scenario with the Ngtcp2 server (Figure 29) we can clearly see when the transfer is interrupted. The third file is a wireshark capture and displays no packets during that period. The other 2 files are log files where we can see attempts at retransmits. We see a couple of flow control frames that the client tries to send, which

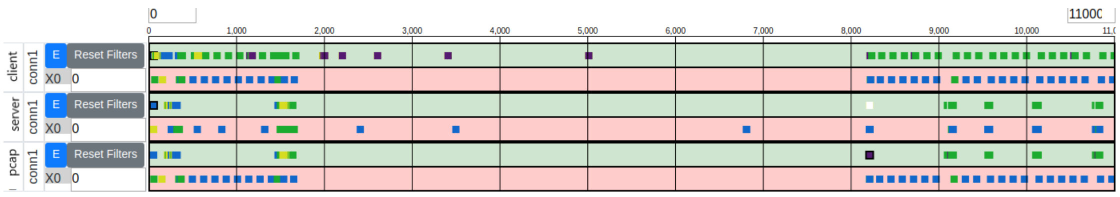


Figure 29: Connection interrupted with ngtcp2 server

are to increase the data offset for a stream. This means the server is approaching the max data limit on stream 4 and the client sends those frames to not block the server from sending more data. When the connection is restored we see that the data transfer is immediately resumed without any abnormalities.

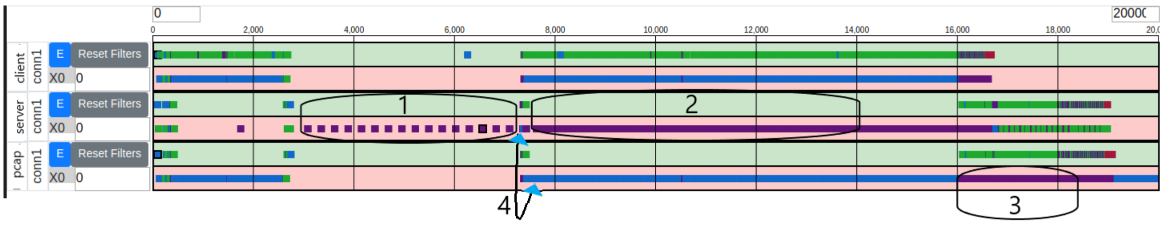


Figure 30: Connection interrupted with quicker server

This is not for all the servers however, when we look at Quicker (Figure 30) we see something entirely different. When the connection is interrupted we can see the server trying to send flow control frames in annotation 1, these are stream_blocked frames. This can happen when the connection is interrupted right before the client updates the max data for that stream. The congestion window of the server is quite big so the amount of data the server sends can quickly go over the max data allowed on stream 4. The client doesn't respond fast enough to pre-emptively send max_stream_data frames.

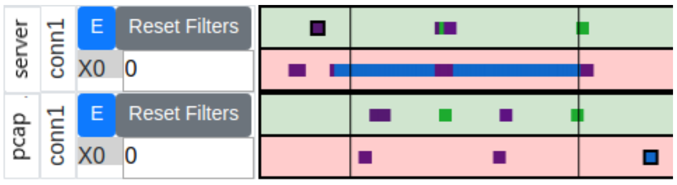


Figure 31: Quicker server flooding connection

When the connection is restored and the server received the max_stream_data frame from the client, the server suddenly does a huge amount of stream data transmits, highlighted in annotation 4. If we zoom in on that, we can see what is on Figure 31. This is way too much for the given bandwidth, to send those packets to the client at the same rate. Immediately after the connection is restored we get a big amount of congestion. The issue with that we can see in annotation 2, where the server hits the max data that it is allowed to send. So the server tries to send blocked frames but, because the connection is so congested those frames are not immediately send. This generates a huge amount of blocked frames which are only sent at annotation 3, which will spam the client with a lot of unnecessary blocked frames. Eventually the server does receive a max_data frame and continues with the data transfer, but not without adding a big delay.

6 Conclusion and Future Work

In this thesis we developed QUIC-Vis, which is a new tool that visualizes data coming from 2 sources to improve debugging and research (Section 4.4.1). With this data, multiple visualisations are generated to help the researchers and developers. When looking at the specifics of QUIC ourselves, we discovered that some things are not so straightforward.

6.1 The weaker sides of QUIC

When analysing QUIC in Section 3, it became clear the this new protocol is very complex. We also see that existing features of TCP are being completely reworked like congestion control. It is good to see this happen, so that we end up with a protocol that can be used for a long time. There is also a big focus on preventing new middleboxes to be developed for QUIC, which would make history repeat itself. Though not all middleboxes are causing issues, QUIC goes in extreme lengths to not give middleboxes any changes of tampering with QUIC.

When QUIC would be introduced, this could cause difficulties with diagnosing network issues. This is because key information is locked behind encryption. This is information like flow control state, retransmissions, congestion control stepping in, ... The only way to unlock this, is by having access to at least one of the endpoints. Where we either need the TLS keys that are used to decrypt a network trace or, a client or server log. It will be interesting to see how much of an impact it would have on troubleshooting network issues. Will it take longer to diagnose and solve network issues and if so, how big of an impact would it have?

If we look at the QUIC specification, more specifically the different frame types that are available. In the latest draft, which is draft 14, there are 20 different types of frames. This can overcomplicate things as a developer, who is trying to implement the logic of all these frames. This also makes researching QUIC quite difficult because there is a lot of information that needs to be understood in order to analyse certain behaviour. If we look at the number of frames to update flow control values, there are 3 of them. Each flow control parameter has its own frame type in order to update the value. Not to mention that for each parameter there is another separate blocked frame, for when there is a flow control issue for that specific parameter. Why don't we simply use 1 frame for updating flow control and 1 for reporting issues. Could we not merge the stream_data and data frames, since stream data is tied to the overall data that is sent/received?

6.2 Designing QUIC-Vis

During development of QUIC-Vis one key aspect always came back up. That is to make the visualisations correctly represent the data, but at the same time make the it easy to read. Since it is a tool designed for researchers and developers of QUIC, the tool needs to be easy to use and show data in a structured way.

This was difficult to do with the sequence diagram. There, the order of arrows need to be sequential so that packet reordering was easy to notice. A problem that we would have is when a few packets were sent around the same time, that would cause an overlap of arrows. In that scenario it would be hard to read what each packet contains in terms of frames, it would also be difficult to see how many packets are being sent. To prevent that from happening, we add a margin to arrows that would be placed to close. By using a 2 pass algorithm, it allows us to place margins where it is needed without reordering any of the arrows.

Something else that we noticed during development was when we were trying to parse server logs of a couple implementations. Looking at Ngtcp2 and Quant we can see a lot more information is available than just packet information. That kind of information (loss detection info, flow control state, transport parameters, ...) will certainly be helpful for doing research or debugging. However, not every implementation supplies that information. For example: Quicker only provides packet information. It would be very helpful to see implementations like quicker to also provide that extra information.

6.3 Using QUIC-Vis

When looking at the test results in the QUIC-Vis tool we could instantly notice some interesting things in the timeline visualisation. For example, when the network is interrupted for a couple of seconds. When opening the related files in QUIC-Vis, we would instantly see a gap where no packets were sent/received. When looking at a Wireshark capture of the same situation, it would be hard to notice at first glance. Also, using colour codes for frame types makes it much easier to find an issue. For example, when using the flow control filter, we could instantly see if a lot of flow control frames were being sent.

The sequence diagram is useful to see the interaction between 2 endpoints. This makes it easier to follow how a connection set-up goes, even for people who have little to no knowledge about QUIC. With the ability to use 2 files, 1 for client and 1 for server, we cannot just see how long the RTT is. But, we can see the time that is needed for the server and client to process a packet and send a response. With this we can instantly determine where the largest delay is.

6.4 Future work

There are still changes being made to QUIC and it will still happen for some time. So a tool like this will be necessary to efficiently to do research and debugging. However, when QUIC is finished and the design is completed, a tool such as this will still be useful for other aspects that are connected to QUIC. One example would be doing research into congestion control algorithms. With TCP and the presence of middleboxes, there is not much space to do research into congestion control. There is not 1 perfect algorithm that will work in every situation. Research needs to be done for the different scenarios, with QUIC allowing algorithms to be “plugged in”, step 1 is covered. For analysing the data, tools such as QUIC-Vis will be needed. The timeline diagram can be used to identify points of congestion. We can look at different times of the connection and see if the amount of packets is lower than usual.

However, our tool is not finished yet, there is still work to be done to improve it. Here are some of the biggest improvements that we want to introduce:

- Highlight packets where protocol violations are being made. This would make debugging a connection easier where a protocol violation happens. Rather than manually analysing each packet, the user can simply highlight those packets with the push of a button.
- Supporting HTTP/2: With QUIC eventually supporting many HTTP/2 features, for example: stream prioritization. It will be important to see if HTTP/2 is supported well by QUIC. With QUIC-vis we could visualize what prioritization each stream has and see if it corresponds with how the streams behave.
- Allow more log files to be supported from other implementations. Due to the limited time available only a subset of implementation logs are supported, this is not enough to fully research the QUIC protocol. Not only that, it will also help the developers with debugging those other implementations.

We hope that this is the start of QUIC-Vis, that it can grow into a well flushed out tool that both researchers and developers can use. These are important times for the internet, we have this opportunity to improve web performance. So for the sake of not repeating history, let us design QUIC in such a way that it becomes a protocol that is very robust and flexible for the future.

References

[1] Ilya Grigorik. “High Performance Browser Networking”. In: (2013). URL: https://hpbn.co/.

[2] Jim Griner et al. *Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations*. RFC 3135. June 2001. DOI: 10.17487/RFC3135. URL: https://rfc-editor.org/rfc/rfc3135.txt.

[3] Jana Iyengar and Ian Swett. *QUIC Loss Detection and Congestion Control*. Internet-Draft draft-ietf-quic-recovery-11. Work in Progress. Internet Engineering Task Force. 30 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-recovery-11.

[4] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-11. Work in Progress. Internet Engineering Task Force. 105 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-11.

[5] Dina Katabi, Mark Handley, and Charlie Rohrs. “Congestion control for high bandwidth-delay product networks”. In: *ACM SIGCOMM computer communication review* 32.4 (2002), pp. 89–102.

[6] Nupur Kothari et al. “Finding Protocol Manipulation Attacks”. In: *SIGCOMM Comput. Commun. Rev.* 41.4 (Aug. 2011), pp. 26–37. ISSN: 0146-4833. DOI: 10.1145/2043164.2018440. URL: http://doi.acm.org/10.1145/2043164.2018440.

[7] Adam Langley et al. “The QUIC Transport Protocol: Design and Internet-Scale Deployment”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’17. Los Angeles, CA, USA: ACM, 2017, pp. 183–196. ISBN: 978-1-4503-4653-5. DOI: 10.1145/3098822.3098842. URL: http://doi.acm.org/10.1145/3098822.3098842.

[8] Daan De Meyer. “HTTP/2 Insight: Visualizing the inner workings of HTTP/2”. In: (2017).

[9] J. Roskind. “QUIC: Design Document and Specification Rationale”. In: (2012). URL: https://goo.gl/eCYF1a.

[10] Jan Rüth. “TCP initial window configurations in the wild”. In: (2018). URL: https://blog.apnic.net/2018/01/15/tcp-initial-window-configurations-wild/.

[11] “SPDY: An experimental protocol for a faster web”. In: (). URL: https://www.chromium.org/spdy/spdy-whitepaper.