De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

Maastricht University

Faculteit Wetenschappen

School voor Informatietechnologie

master in de informatica

Masterthesis

Quic congestion control and loss recovery evaluation

Jonas Reynders

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Peter QUAX

2019
2020

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Quic congestion control and loss recovery evaluation

Jonas Reynders

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Peter QUAX

Masterproef: QUIC congestion control and loss recovery evaluation

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

2020

A thesis presented for the degree of Master of Computer Science

Abstract

QUIC is a new transport protocol that is currently being developed by the QUIC working group. QUIC is built on top of UDP and will be integrating TLS and parts of HTTP/2. It has been in development for several years now and version one should be finished soon. Part of the standardisation process is congestion control and loss recovery logic. Rather than designing a new congestion controller, the working group decided to build on existing algorithms with some changes. In this thesis we will explore these changes and compare them to TCP congestion control and loss recovery. We will dive into the code and use network simulators to assess the performance of the new upcoming standard. However, evaluating congestion control and loss recovery is quite a challenge with how networks are being simulated.

Acknowledgements

First and foremost I would like to thank my mentor Robin Marx. Throughout the thesis, his expertise on QUIC and current congestion control algorithms has helped me with getting a better understanding of the complexity of this subject.

I would also 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

Even though it feels like the internet has been a part of society for a long time, relatively to other inventions it is very young. However, over the years it has changed quite a bit and is now used for all sorts of goals. However in some areas the internet has been trying to catch up.

One big component that is used for sending data all over the world is the transport protocol TCP, which is still in use today. It was designed in order to send data over the IP protocol in a reliable way, making sure that data arrives and in the correct order. Initially it achieved its goal of sending data over a network from one computer to the other.

However, not long after TCP was deployed, an issue was discovered. The great congestion collapse happened in 1986, where a network was so overload that all the computers part of it were struggling with retransmitting lost data. This problem made data transfer incredibly long and so, a solution was developed to deal with it.

Ever since 1988, congestion controllers were a part of TCP and made sure that data was being sent without overloading the network too much. By limiting the send rate and listening to congestion signal, it provided a great benefit for sending data over the internet. As recently as 2018, new congestion control and loss recovery algorithms are being developed, where each new algorithm tries to fix issues previous algorithms have.

Over time issues with the TCP protocol itself were discovered and so a new transport protocol is being standardised. QUIC is the name of this new protocol which will fix many of the problems discovered within TCP. With QUIC also being a reliable transport protocol as TCP is, it will also need congestion control and loss recovery algorithms to work efficiently. With QUIC supporting a different structure for sending data, re-using a congestion controller from TCP is difficult to do, as well as that the extra information in QUIC would be unused.

The work group for QUIC decided to modify an existing congestion control algorithm, NewReno, along with newer loss recovery mechanisms rather than complete start over. Designing a whole new transport protocol is already a challenge, so it was decided that designing a whole new congestion controller is out of the scope. The idea being that the modified congestion controller can be used as baseline for implementations, that should improve performance compared to TCP.

However, NewReno is quite an old algorithm and so is it worth using, even if there are changes made which should improve it? In this thesis we will try to evaluate this new congestion controller and compare it to a TCP implementation to see if these modifications provide a solid increase in performance. We will also explore the current algorithms that are available for TCP and analyse how they work differently conceptually. This should give us more insight in congestion control and how difficult it is, since new algorithms are still appearing.

2 QUIC

With QUIC being developed on top of UDP, many aspects of TCP have to be implemented again. Given this, it is a great opportunity to redesign flow control, congestion control, security, ... Currently the QUIC protocol is on draft version 29[1] and is still being updated. Since we will be focusing on the congestion control and loss recovery, we will first highlight the largest changes in QUIC compared to TCP that will affect congestion control.

2.1 Connection ID and Connection migration

QUIC uses a different method to identify connections by using connection IDs, rather than relying on the IP/PORT tuple. Due to using connections IDs to distinguish different connections, it is possible to keep a connection alive when endpoints change IP address and/or port. After a new path has been validated, data can be transmitted to the endpoint again without the need of creating a new connection. This new path however may not have the same bandwidth available and thus the congestion controller may need to be reset.

With the use of connection IDs it is also possible to extend QUIC to allow for multiple network paths to be used at the same time. An example would be with a smartphone where data can be sent via WiFi and mobile internet simultaneously. This however makes it more challenging to deal with congestion control. Each path can be different in terms of maximum throughput, delay, amount of competing connections, ... The challenge would be to optimally uses both paths whilst managing congestion on both paths. However, we will be focusing on the single path congestion control and loss recovery in this thesis.

2.2 packet numbers

The usage and generation of packet numbers has changed compared to TCP. QUIC uses packet numbers purely to identify the order of packets and separate different packets. TCP's sequence numbers were also determined based on the offset for the data (Figure 1), making sure the receiver knows in what order the data needs to be sent to the application. QUIC increases the packet number by one for each new packet, the data offset will be included in the payload. When dealing with re-transmissions QUIC will not resend the lost packet like in TCP, it will create a new packet with a new packet number and place the lost payload into the new packet. This helps with reducing ambiguity when an acknowledgement is received for a packet, with QUIC it can easily be identified if a re-transmission or the original packet is acknowledged. This makes it also easier to detect spurious re-transmissions, allowing the congestion controller to change some thresholds to detect loss less aggressively.

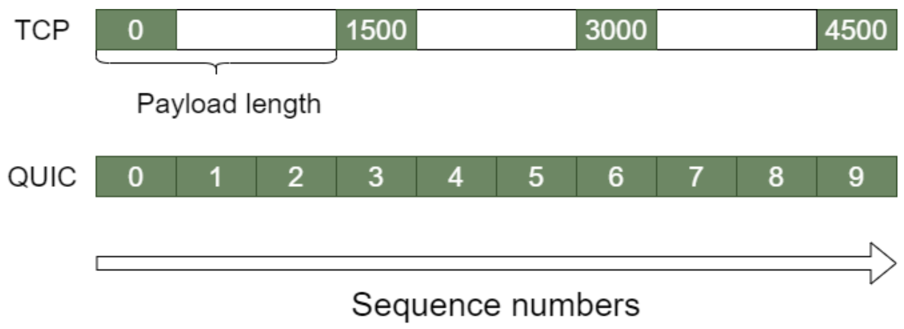


Figure 1: Generation of packet numbers in QUIC compared to TCP

Rather than using a single packet space for the entire connection (similar to TCP), QUIC uses three different packet number spaces: initial space, handshake space and application data space. Each space can have a counter for the packet number, thus each space can start from packet number 0. Due to how packet numbers are maintained as mentioned previously, a single counter can be used as well and even gaps can be introduced in the packet numbers.

The initial and handshake spaces are used temporarily during connection setup, eventually only the application data space will be in use. With TLS being integrated more closely than in TCP, there were complex issues when finishing the crypto handshake and switching to encrypted packets[1]. An example is when the client is finished with the crypto handshake and sends a client_finished message[2], this packet would not be encrypted yet because the server might not be finished yet. It could be that this packet is lost and might needed to be re-transmitted with a higher packer number. Since it is also possible that the acknowledgement could have been lost, the server will respond with an acknowledgement. However, with the packet not being encrypted, it is possible to execute a replay attack. A third party could send another client_finished packet whilst increasing the packet number to the server. If the server responds to the first client_finished message, it will increase its counter for the packet number. This packet number will then be used to send data, this is the same packet number as the third party used for the replay attack. The server will now ignore the data packet because it has already seen a packet with that sequence number. The client would also receive an acknowledgement for the replayed packet, which it thinks is an acknowledgement for the data packet.

Separating the packet number spaces also helps with preventing spurious re-transmissions for application data when loss occurs for handshake packets[3]. QUIC allows endpoints to send data immediately without having to wait for a handshake to be completed, this can only be done if the client has connected to the server before and stored the transport

[1] https://docs.google.com/presentation/d/176bVI27bRJrfahf8RR89hbZ_owMs54ipxXALNUUsV1c/edit#slide=id.g3baba8f64c_0_202

[2] https://github.com/quicwg/base-drafts/issues/1018

[3] https://github.com/quicwg/base-drafts/issues/1413

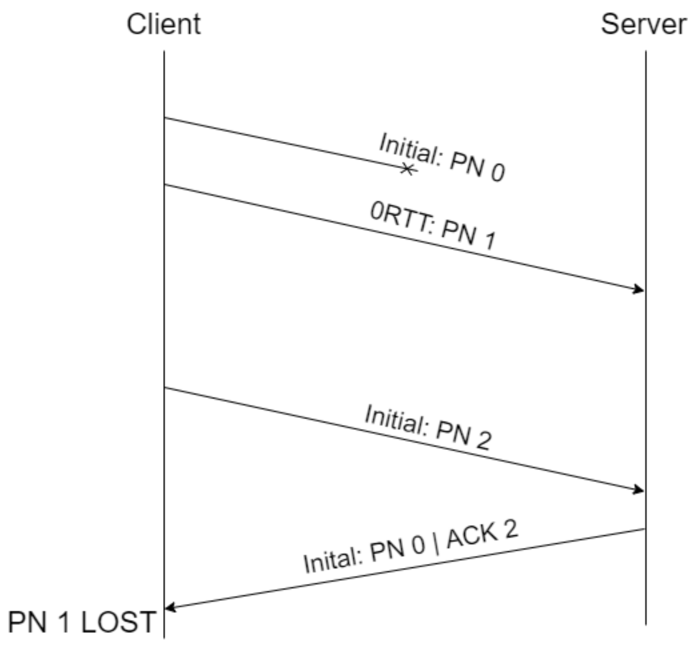


Figure 2: Spurious loss detection with 1 packet space during 0RTT

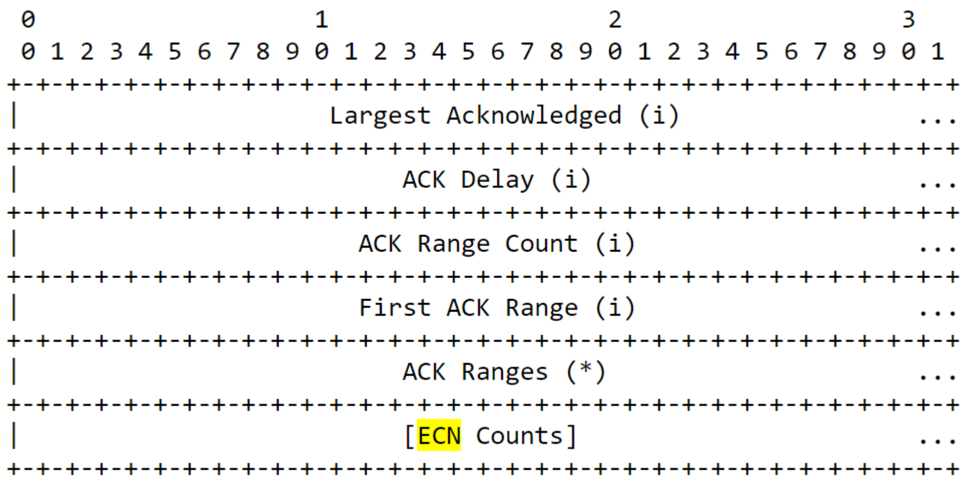
parameters. This reduces the amount of roundtrips needed to retrieve a small amount of data. In Figure 2 we can see an example of a 0RTT connection setup when using one packet number space. Let us assume that the first initial packet is lost, which is similar to the SYN packet TCP uses. The second packet, with packet number 1, arrives as expected. However, the server will only acknowledge 0RTT packets once the handshake is completed since it contains encrypted data. Since the server doesn't send an acknowledgement at all, a timer will fire and the client sends another initial packet that does arrive. The server will respond with an acknowledgement this time, but will only include packet number 2 since the server is not ready yet with the encryption keys. Once the client receives the ack, it can detect loss due to ack-based loss detection which we will describe in greater detail later on. The client will now send the 0RTT data again, even though the first 0RTT packet was received.

2.3 Packet structure

The QUIC packet structure is also significantly different from TCP, both header and payload structure are different. First, there are two different header formats used during a connection. When establishing a connection the long header variant is used. When exchanging cryptographic and transport parameters is finished and the connection has been established, QUIC will switch to the short header variant to reduce overhead. The payload of packets will be divided up into frames, where each frame has a specific purpose.

This is quite different from TCP where only one header type is used and the payload will only contain data.

Compared to TCP, certain header information was removed from the QUIC header: flow control, congestion control, error reporting, ... This reduces overhead, for example when a server that is only sending data, it does not need to continuously advertise its available flow control window. Also flags which are only one bit in size were removed, this does make an impact for long living connections. For QUIC this functionality was moved to the payload, where the payload will consist of specific frames.



```
 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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Largest Acknowledged (i)                ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          ACK Delay (i)                      ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       ACK Range Count (i)                   ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       First ACK Range (i)                   ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          ACK Ranges (*)                     ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          [ECN Counts]                       ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3: Ack frame structure [1]

There are many different frames that are defined and each has its specific use, e.g. stream frames are exclusively used for sending data. An important frame we will be looking at is the ACK frame (Figure 3), which will report which packets were acknowledged. Starting from the basics the ACK frame contains what the largest acknowledged packet number is. This can be used for accurate RTT measurements, where an endpoint waited the least amount of time to send an ACK which gives a better picture of the network delay. Similar to TCP Selective acks[2], QUIC also allows multiple ranges of acknowledged packets when gaps are formed because of packet loss. The maximum amount of ack ranges TCP can have is three due to the limited header size. QUIC can support more than three ranges, allowing it to notify the sender of all the packets that have been received. In a very lossy environment, e.g. every third packet lost (Figure 4), the amount of ack ranges can easily go over three. As explain earlier re-transmissions are given a different packet number, so the gap in the ACK frame can stay there for a while and having a limit of three ack ranges can prevent the sender from re-transmitting more data.

QUIC does not allow reneging, which happens in TCP when packets that were acknowledged through SACKs are dropped from the SACK header and are considered now lost for the other endpoint. This allows TCP to safe buffer space since it can't send those

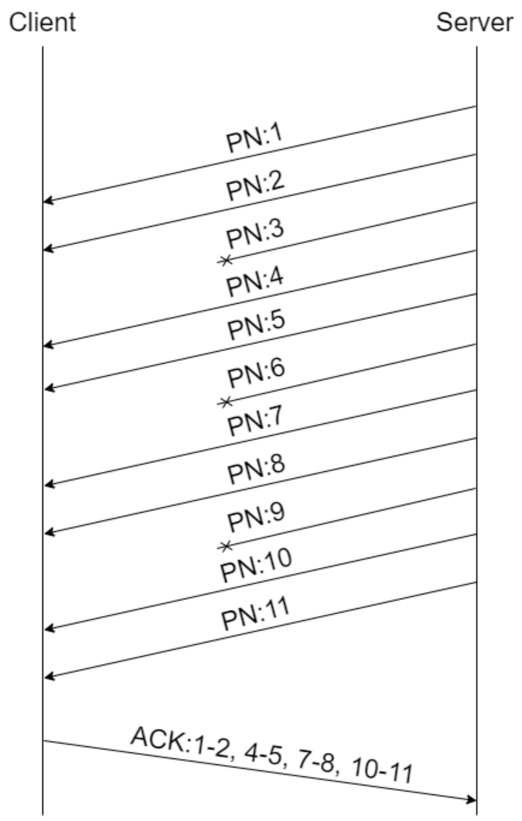


Figure 4: Use of ack ranges in lossy environment

packets up to the application layer. QUIC decides to not allow reneging to not complicate things for the sender, allowing it to clear up buffer space by removing the acked packets.

In the ACK frame there is also a field called *ACK Delay*, which indicates the time difference in when this ack was sent and when the largest acknowledged packet number was received. In Figure 5 we can see the ack delay visualised, compared to it we can also see that the measured RTT on the server includes this delay. Knowing the ack delay allows for more accurate RTT measurements, where the time lost for generating an acknowledgement can be subtracted from the total delay measured. TCP doesn't measure the ack delay, thus it increases threshold values for loss detection to prevent spurious re-transmissions for delayed acknowledgements.

In the ACK frame we can also see an ECN Counts field, to support Explicit Congestion Notification[3]. Let us first give an introduction to ECN and how it is used with TCP, we will go into further detail about ECN in Section 3.2.2. ECN starts in the IP layer where there are two bits available. ECN is present on the IP layer because congestion can build up at any node in a network, regardless of which protocol is used above IP. Before ECN can be used with TCP, the endpoints first need to see if all the nodes along the path support ECN. Once that has been established, a node can notify the receiving

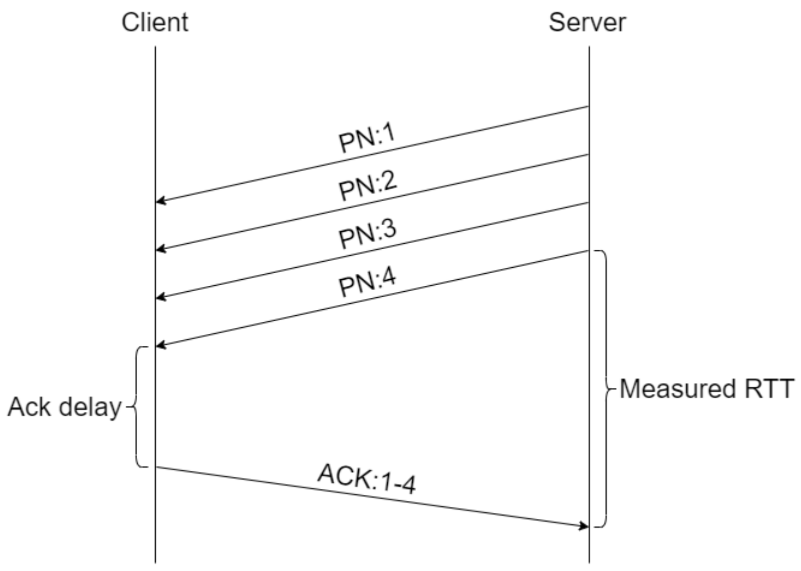


Figure 5: Ack delay visualised in contrast to measured RTT

endpoint that there is congestion building up. However, the receiving endpoint knows that there is congestion but the sender (who is causing congestion) does not know it yet. So the congestion notification needs to be relayed back to the sender.

Notifying the sender of congestion is done via TCP and not IP. This is because that TCP is sending the data and has a congestion controller that determines how much data can be send. In order to make sure the TCP congestion controller knows about the congestion, TCP has two bits in the header: ECN-echo and CWR. The Echo bit is used by the receiver to indicate it has received packets where in the IP header congestion was detected by the ECT and CE bits. When the TCP sender receives a TCP packet with the Echo bit set, it will slow down with sending data. The CWR bit will then be set by the sender to notify the receiver that it has received the congestion notification and is reacting to it. The receiver can then stop setting the Echo bit high, until congestion is detected in the IP layer again.

In QUIC there is the ECN COUNTS field which includes three values: ECT0 count, ECT1 count and ECN-CE count. ECT0 and ECT1 counts have the same meaning, these two values on the IP layer are represented as 01 or 10 for the two ECN bits. Both values can be used the signal that the connection is ECN capable. The ECN-CE parameter keeps count how many IP packets were received with the two ECN bits having a value of 11, otherwise the ECT0 or ECT1 count parameters is increased. With QUIC allowing delayed-acks as described earlier, this can give the sender a better indication on how much congestion there is. When the ECN-CE counter is very high that means there is a lot of congestion, this information can then be used to determine how much the sender should reduce its send rate. By having the counters also allows QUIC to see how congested the network is over time, if the ECN-CE counter increases or decreases.

Next are a set of frames that deal with managing flow control, which next to congestion control can also limit the amount of data that can be sent and amount of streams that can be created. QUIC uses a credit-based flow-control method where an endpoint sets a limit on the amount of total bytes it can receive (or streams it can open) during the connection. TCP uses a sliding window that has a maximum size, the window starts at the last acknowledged packet through the default acknowledgement header field. With flow control information not being included in the header anymore, a credit-based method requires the least amount of updates. It would not be necessary for each acknowledgement to also include what the current flow control window is. Initial values are sent during the handshake whilst during the connection a set of flow control frames are sent to update this value. With QUIC having streams in play, there are two levels of flow control for receiving data: connection-level and stream-level flow control. The amount of data sent cannot go over the connection-level limit, even if that specific stream allows more data. This is to prevent the receiver being overflowed with data.

2.4 Congestion control

The idea for QUIC is to be a better alternative to TCP, this section highlighted a few improvements QUIC has made compared to TCP. However, this is mostly about the protocol itself and how it will be sent over the wire. QUIC, similar to TCP, wants to guarantee in-order delivery of data. It also makes sure that all the data is eventually received by the other endpoint. The mechanisms are there in the protocol (e.g acknowledgements). QUIC will also need to detect and re-transmit lost packets as well as not overload the network by sending too much data at once. In the next section we will describe what happened in TCP over the past thirty years and how these challenges were handled.

3 Congestion Control and loss recovery

The TCP protocol is designed to deliver data in the correct order and make sure all the data is delivered to the other endpoint. It also wants to do it as fast as possible. This is quite the challenge as we will discover in this section.

First there is the possibility of data being lost. Packets can become corrupted due to bit values changing when it is being sent on the wire or wireless, this can happen due to noise. Corrupted packets are being dropped because of the contents of the data being changed. Since TCP is designed to deliver all the data, this lost data will need to be retransmitted. With in-order delivery being important as well, the lost data should be retransmitted as fast as possible so the receiver can deliver the data in the correct order to the application layer.

Another challenge is being able to send data fast over the internet. We can represent the internet as a collection of nodes in a graph where links between nodes allows the transfer of packets. The problem is that each node can process packets at a certain rate, not to mention that each link also has a maximum transmission speed for data. We will refer to this send rate as the bandwidth, which expresses the maximum amount of bytes can be sent per second.

A problem arises where for one node packets arrive at a faster rate than what can be sent out. When this happens, packets that arrive whilst another is being processed will be dropped. To prevent this from happening, buffers were placed to store packets that arrive but cannot be processed yet. However, this just delays the dropping of packets, since buffers have a limited size. When this behaviour occurs, it is called congestion.

In 1986 congestion happened on a large scale[4], where it was discovered that the send rate of data, dropped drastically for active connections. The rate at which an endpoint sends data, we will refer to it as the throughput. Researchers saw a drastic drop in throughput and when inspecting why it was happening, they noticed that a lot of data was being retransmitted. What happened was that the buffers over the network path, were filling up and thus causing other packets that arrived to be dropped. This causes TCP to retransmit data but with the network being in a congested state, those retransmissions would be dropped again.

So researchers were faced with the issues of congestion and the effect it had on transmitting data. Something had be done about it in order to allow data to be transmitted in a smooth manner. In this section we will go over the different solutions that were proposed and how these have evolved over time over a span of thirty years.

3.1 First methods to reduce congestion and recover lost packets

Let us start first with how loss can be detected with TCP. As mentioned in the previous section, TCP uses an acknowledgement method to signal the sender what data has been successfully received. However, packet loss can happen on any connection and TCP needs to know which are lost so it can retransmit them. The sequence number in the acknowledgement is the sequence number of the packet that is the next one in order. Because of this, when a data packet is lost, the receiver will continue to send acknowledgements of the same sequence number. A first loss detection method is to observe if acknowledgements have the same sequence number, this is called the duplicate ack method. If TCP notices that three duplicate acks are received, the packet will be assumed lost.

A threshold of three is used to reduce the amount of unnecessary retransmissions, or spurious retransmissions, due to packets being reordered. Reordering can happen if multiple paths exists and one path might experience congestion, new packets can than be sent over the other path. That other path might have less packets in the buffers and thus packets do not need to wait as long than in the first path, causing packets that were sent later to arrive earlier. In this case, waiting a bit before declaring a packet lost can prevent spurious retransmissions since duplicate acks will be received.

A second method of detecting lost packets is with the use of a timer[5]. A problem that can occur in the first method is when one or more of the final packets are sent and the sender does not have any more data to send. It can happen that the receiver does not see any packets arriving or maybe just two packets with a higher sequence number. The receiver would then not send any acks or less than three duplicate acks. This does not the trigger the duplicate ack method. A solution was then to use a timer, when a packet would not be received within a certain time, the timer would expire and the packet would be marked as lost. The timer would be dynamically calculated based on the experienced delay, to prevent loss being detect to early on high delay networks. The timer value was restricted between the value of one second and one minute, the lower bound was chosen because of the old systems not having clocks that were accurate enough for a lower value.

As described earlier loss can happen due to the network being in a congested state. This can happen when more packets are arriving at a node than can be sent away. A solution to this is to limit the sender on how much data it can send at a given time. So was the Congestion Window (CWND) introduced, which limits the sender. The CWND contains the amount of packets that can be sent over the network without needing to be acknowledged. Once a packet is acknowledged, a new one can be send. This will limit the throughput of the sender, which does not go over the maximum available bandwidth available.

The challenge is now to manage the size of the CWND to fully utilise the available bandwidth without causing another congestion collapse. What makes it more difficult is that there can be more than one TCP connection on the same path or other types of traffic like UDP. If two TCP connections are sending at a throughput that is equal to the maximum available bandwidth, the total throughput would be equal to two times the bandwidth. This would cause congestion again, so connections need to be fair towards each other and share the available bandwidth.

When a TCP connection is started, the state of the network is unknown. Thus a method was needed to probe for the available bandwidth. A first method is called slow start[4], which starts at a small congestion window of one packet. The sender sends that amount of data and waits for an acknowledgement. Once an acknowledgement is received, which means the data was not dropped due to congestion, the sender will increase the CWND by one packet. This means the sender can now send two packets because the acknowledged packet is not counted towards the CWND limit anymore. Essentially the CWND will double for every round trip.

Another method is called congestion avoidance[4], which as the name suggests wants to avoid causing congestion. With this method the CWND will only increase by one packet if all the packets that were in-flight, with the previous CWND size, have arrived at the receiver. This causes a much slower growth of the CWND than slow start, but slow start can increase in size very fast the longer it is running. A sudden increase in the CWND can cause congestion, which in turn can cause a lot of lost packets. With congestion avoidance it takes longer for congestion to form and when packets get lost due to congestion it will not be that many.

Since both methods will cause congestion at some point, packets will be lost. With the loss detection methods described earlier, lost packets can be identified. It is important to send these packets back as fast as possible. For that TCP can switch to the recovery phase, where lost packets will be sent again. During this phase no new data will be sent, because the TCP receiver needs to have space available to receive the lost packets. The application layer expects data to be delivered in the correct order and so the packets that were received with a higher sequence number need to wait. Once the retransmissions are received by the other endpoint, TCP can go back to sending new data. With packet loss happening with the current congestion window, it means that resulting throughput of the current CWND size is too large. So, the CWND needs to be reduced to one packet in slow start and half its original size in congestion avoidance. From there either of the two methods described earlier can be repeated to find a new maximum CWND size.

3.2 Evolution of congestion control

Since 1988 congestion controllers have started to appear in TCP implementations, which manage the CWND size. These congestion controllers work together with loss recovery to send data as fast as possible without causing another congestion collapse and guaranteeing that all data will be received. We will go over many different congestion controllers and see how they have evolved over the years.

3.2.1 Introduction of loss based algorithms

One of the first algorithms that was introduced, is TCP Tahoe[4] (1988). Slow start, congestion avoidance, loss recovery concepts were already known but have not been combined into one algorithm until TCP Tahoe. However, this algorithm was very basic: resetting the CWND to one packet after experiencing any kind of loss, when a timeout happened. Tahoe will first enter slow-start to be able to grow the CWND fast. Once packet loss is detected and the recovery phase ended, the CWND will be reduced and slow start will be performed again. The current CWND size divided by two will be kept in the SSthresh variable. Once the CWND grows past the SSthresh, the algorithm will switch to congestion avoidance. Since slow start increases the CWND exponentially, it can suddenly cause a lot of congestion and loss. By switching to congestion avoidance, TCP can send more data at stable rate and when packet loss occurs it does not need to stay long in recovery. The CWND in Tahoe would follow an Additive-Increase/Multiplicative-Decrease (AIMD) flow.

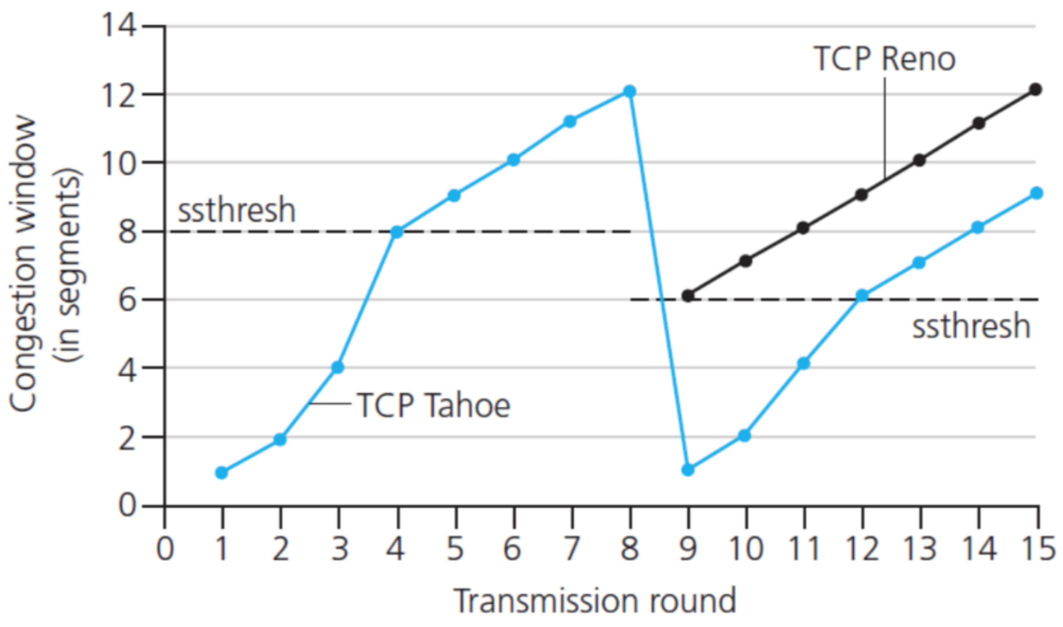


Figure 6: CWND behaviour in Tahoe compared to Reno source: https://www.chegg.com

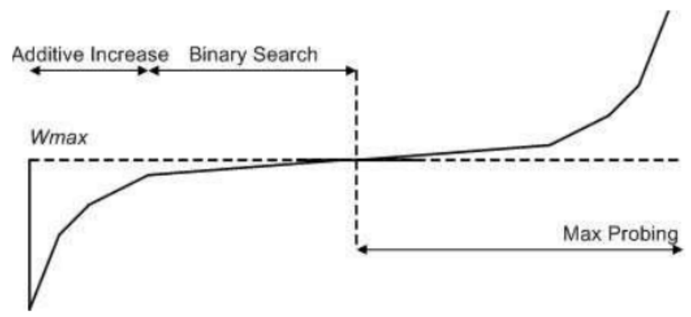
Later, TCP Reno[6] (1990) was introduced which added the concept of fast recovery to TCP Tahoe. A problem that Tahoe had was when a single packet was lost, this would cause a CWND reset. This packet could have been lost due to corruption and thus resetting the CWND would under utilise the available bandwidth by a large margin. So Reno added a Fast recovery method when detecting loss via duplicate acks. The CWND would be halved after such a loss event (Figure 6) and the algorithm would immediately continue with congestion avoidance. It also uses additional duplicate acks during recovery to send more data since less packets are in-flight, so additional retransmissions are not blocked until recovery is done.

Reno can exit recovery too early when it receives a new ack. Some packets may not get retransmitted during Reno recovery, which can happen if the lost packets are not grouped together. An example could be where packets with sequence numbers 11 and 20 are lost. First packet 11 will be detect as lost, so Reno goes into recovery. Packet 11 will be retransmitted and acknowledged, but the sender gets an acknowledgement for packet 19. Since new data has been acknowledged, Reno goes out of recovery and starts sending new data after halving the CWND. However, now duplicate acks will be received for packet 20, thus Reno goes into recovery and reduces its CWND again. TCP NewReno[6] (1995) solves this by staying in recovery until all previous in-flight data has been acknowledged.

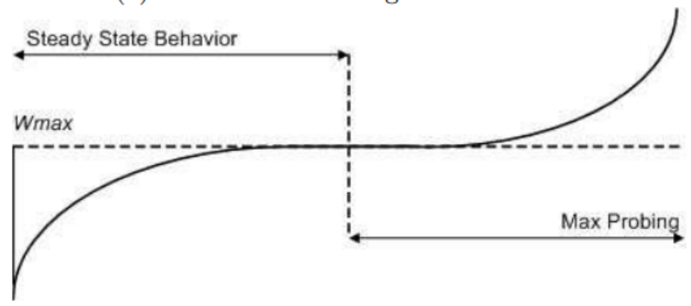
The previous algorithms all have the same additive-increase/multiplicative-decrease functions for the CWND. It can occur that due to loss the CWND is reduced by so much that the resulting throughput is quite low compared to the available bandwidth. This will mean that the link will be under utilised for quite some time since the CWND grows by one segment per RTT. Another consequence of letting the CWND grow per RTT, is that a connection with a lower RTT will grow faster and thus take more resources. Both the slow-start and congestion avoidance methods increase their CWND when acks are received. Slow-start does it when one packet is acknowledged and congestion avoidance does it when a CWND amount of packets are acknowledged. This means that the lower the delay is and the faster acks arrive, the faster the CWND grows. A connection that has a longer delay, will have a CWND that grows slower and will not be able to send as much data as the connection with a shorter delay.

TCP BIC[7] (2004) uses a binary search function in congestion avoidance, and goes through three different phases in congestion avoidance. First it has a few thresholds: W_{min} which is the CWND size where no loss has occurred, W_{max} which is the CWND where loss has occurred and detected by duplicate acks, S_{min} which is the minimum increase in CWND and S_{max} which is the maximum increase in CWND. In Figure 7 we can see the three different phases: Additive increase, binary search and max probing. First Bic goes through additive increase, here the CWND will be increased by S_{max} till it reaches W_{min}. It does not go immediately to W_{min}, because a large increased could cause congestion due to the burst of packets that will be sent at one time. Once it reaches W_{min}, it switches to binary search where the increase in CWND will be $S_{min} < increase < S_{max}$. This phase allows other connections with higher RTTs to catch up. Once the CWND is past W_{max}, it switches over to max probing. Here the increase of CWND starts at S_{min} and grows to S_{max}, this is done to not cause a large amount of congestion if the available bandwidth has grown only by a small amount. In case the available bandwidth has grown a lot, BIC will reach it faster than New Reno.

The different phases in BIC are quite complex to implement and since the algorithm was published, a new algorithm was developed. TCP CUBIC[8] (2008) follows the logarithmic growth of the congestion window by using a cubic function. BIC also is not fair towards other algorithms when the RTT is very low. Even though the CWND growth slows down in binary search, due to a low RTT it can get past this phase fast as well as the beginning of the max probing phase.



(a) BIC-TCP window growth function.



(b) CUBIC window growth function.

Figure 7: CWND behaviour in BIC compared to CUBIC[8]

TCP Cubic was introduced to reduce the complexity of BIC by using a cubic function which allows the CWND to grow with the same pattern as BIC. When loss is detected, the value of the CWND will be stored in W_{max} which is where the cubic function will slow down in growth. The CWND will be reduced by 20%if loss is detected and from there the CWND can grow again using the following equation: $W(t) = C(t - K)^3 + W_{max}$. The parameter C is the cubic parameter which has the value of 0.4, this determines how long Cubic slows down when the CWND is close to W_{max} to allow other connection grow their CWND as well. The parameter t is the time elapsed since the last CWND reduction and K is the time needed to grow the CWND to reach W_{max}. The time period K is calculated by using the cubic parameter and loss reduction factor, where the time increases if the loss reduction factor is larger.

A downside loss-based algorithms have, is that packets will be lost since they use it as a signal for congestion. This means that first the algorithms have to go into a recovery period, to retransmit the lost data before new data can be sent. Having a different signal without needing to recover lost packets, could reduce the time lost when the algorithms have to go into recovery. With TCP also using cumulative acks, it is hard to know if multiple packets were lost.

3.2.2 Extentions to TCP and IP

Very early on a problem was spotted when dealing with retransmissions due to TCP using cumulative acknowledgements. The use of cumulative acks made it easy to acknowledge a range of packets by sending an ack for the largest sequence number. However when dealing with loss, duplicate acks would be sent, but in the mean time additional packets with higher sequence numbers could be received but not get acknowledged. This meant that after successfully sending a retransmission, TCP would be unsure if additional packets were lost as well. TCP got an extension by the name of SACK[2] (1996) or Selective acknowledgement. This allows TCP to acknowledge packets with higher sequence numbers that have arrived before the lost packet, it would also highlight additional lost packets if they are excluded from SACK.

With the use of SACKS a new loss detection method (2003) could be used as well. Rather than waiting for the amount of duplicate acks to go over a certain threshold, TCP could look into the SACK ranges to see how many packets were acked. Since each new arriving packet triggers a duplicate ack, the same threshold could be used when counting SACKed packets. Let us say that packet 2 has been lost and packet 3 to 5 arrive. For each packet, TCP would generate a duplicate ack for sequence number 2. With SACKs, the sack range would be for packets 3 - 5 when sequence number 5 arrived. This sack range acknowledges three packets that were sent after sequence number 2. Since the amount of duplicate acks is equal to the sack range, loss can be detected by looking the size of the sack range. This has the additional advantage when duplicate acks packets would be dropped, to instantly go into fast recovery on receiving the first duplicate ack with more than two SACKed packets.

Another approach was suggested when dealing with congestion buildup in the nodes that partake in the network path. When every link from the sender to the receiver has the same bandwidth or the link speed increases for each subsequent link, congestion will build up first at the first link where the bandwidth is the lowest (Figure 8). What is expected that when a queue is full, packets that arrive will then be dropped which serve as a signal for congestion. However, the first loss will be detected after those queued packets were received by the other endpoint. This allows the sender to continue send data as acks are still being send by the receiver, which continues to cause congestion. With the use of Active Queue Management, nodes can drop packets before the queue is full. This triggers the duplicate acks earlier than previously, preventing the sender of filling up the queue completely.

With AQM, loss will be still experienced and those lost packets will have to be retransmitted. Ideally the node can just send a signal whilst it is forwarding packets, that signal can then be relayed to the sender which is causing congestion. Explicition Congestion Notification[3] (1994) introduces a method to signal congestion buildup to TCP endpoints. ECN bits are both present in IP and TCP as explained in Section 2.3. First the TCP endpoints negotiate to see if both support ECN, this is done by setting the both the ECE and CWR bits. The other endpoint will respond by only setting the ECE

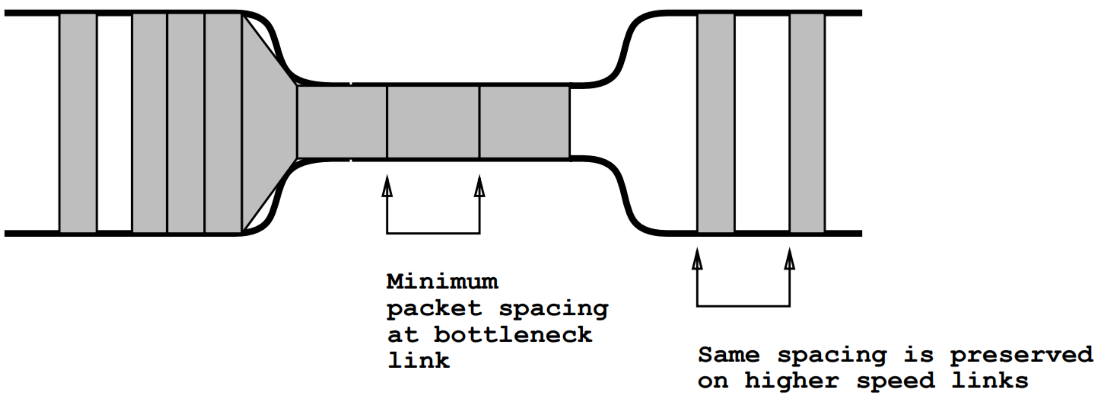

Minimum packet spacing at bottleneck link
Same spacing is preserved on higher speed links

Figure 8: Queue filling up a link with lowest bandwidth[9]

flag, if it supports ECN. After the setup in TCP, the sender of data can enable the ECT bit in the IP header. This allows nodes in the network to set the CE bit, if it supports ECN and see a build up of congestion in the buffer. Looking at an example in Figure 9, when a node observers that a buffer is filling up it will signal it via the IP header bits. Each subsequent node will either forward it or drop the packet. If the TCP destination receives packets with the IP header signalling congestion, it will then forward it via the TCP header to the sender to signal the congestion controller to reduce the CWND. Since the sender is causing the congestion, it needs to notified.

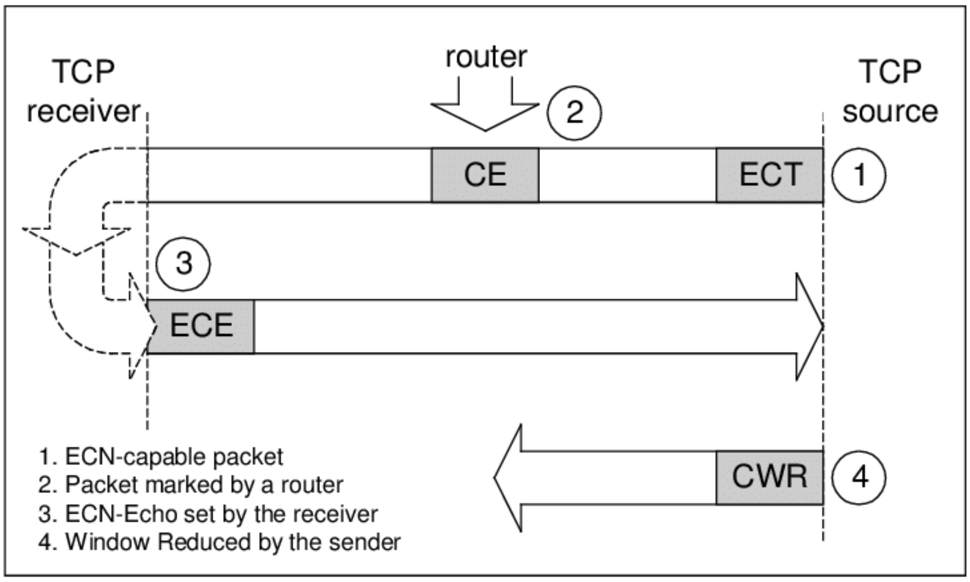

TCP receiver
router
TCP source
2
CE
ECT
1
3
ECE
CWR
4
1. ECN-capable packet
2. Packet marked by a router
3. ECN-Echo set by the receiver
4. Window Reduced by the sender

Figure 9: Example of ECN-based congestion detection [10]

With the addition of ECN an interesting use case was introduced to use ECN as a primary signal for congestion in data centers. Since the network environment is controlled within a data center, ECN capable nodes can easily configured. A new Congestion Control algorithm was proposed to use ECN as a primary congestion signal, which was named Data Center TCP (DCTCP[11], 2010). When having a shallow buffer on a network interface, large amounts of packet loss could be caused when data is sent from multiple nodes at the same time. When each node receives a notification of packet loss, they will all retransmit at the same time and cause the same problem to be repeated. Additionally when using loss-based algorithms for a long time, buffers would be filled up leaving not much room for additional connections. This is due to loss being the signal of congestion and packet loss happening when the buffers are full. Even if AQM is used, the buffers will still be slightly filled.

To prevent this, DCTCP aims to utilise the available bandwidth whilst keeping queuing delay to a minimum, by using ECN to signal not just the buildup of congestion but also the extent of filled buffers. By counting the amount of ECE flagged packets, each endpoint could estimate the amount of congestion and reduce its CWND by that same extend. This means that DCTCP can reduce congestion very quickly when needed and also not so much that the bandwidth would be underutilised. Both slow-start and congestion avoidance used in New Reno are used to grow the congestion window. DCTCP will exit slow-start when it first receives a packet with the ECE flag enabled, then it switches to congestion avoidance for the rest of the connection.

3.2.3 Using delay in congestion control

A big problem all previous loss-based algorithms have, is that they purely rely on the packet loss signal to detect congestion. With how networks are built up, buffers will already be full when this happens. Due to how loss-based algorithms increase the CWND, network buffers will be filled up even if the maximum bandwidth is reached. These loss-based algorithms basically introduce congestion themselves in order to detect it.

In 1979[13] an optimal operation point was discovered where the throughput equals the maximum bandwidth and where the packets do not need wait in the queue. In Figure 10 we can see where that point is in terms of the delivery rate and the measured RTT. A side effect of placing packets into buffers, is that an extra delay is added (queuing delay). This delay will increase when buffers are being filled up more and more. When the throughput is lower than the available bandwidth and the buffers are empty, there won't be any queuing delay. Loss-based approaches operate around the point of maximum queuing delay, where the queue is full and packets are being dropped. It was discovered at the same time that creating an algorithm that operates at the optimal point was incredibly difficult. However, an approach that tries to achieve a close enough result was discovered by measuring the build up of queuing delay.

TCP Dual[15] (1992) is the algorithm that first introduced the concept of using RTT measurements to decide the growth of the congestion window. With loss-based algorithms you also have an oscillation happening with the CWND due to the AIMD method. In

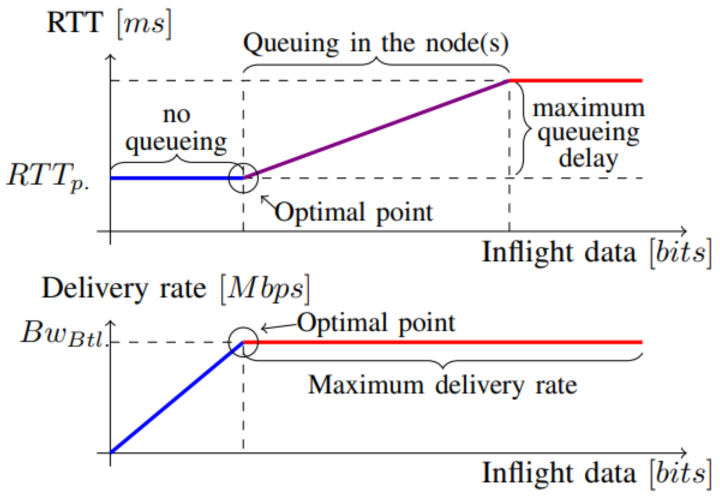


Figure 10: Optimal point to maximise throughput and keep queuing delay low [12]

Figure 11 we can see an example of how the CWND changes over time when using Reno as the congestion controller. We see that the CWND follows a saw-tooth pattern, the CWND increases linearly over time in congestion avoidance and is reduced by half when loss is experienced. Due to loss happening when a queue is full, the larger the queue the longer the CWND can grow. When the CWND can grow to a large size, it changes by a large margin when loss is detected.

With the use of RTT values, congestion can be detected early and thus also not require large changes to the CWND. TCP Dual will make use of two RTT constants: RTT_{min} which is the minimum transmission delay and RTT_{MAX} is the RTT when high congestion is present and which also includes queuing delay. With these two values and the latest RTT value, the current and max queuing delay can be calculated. If the current queuing delay values goes over the max queuing delay threshold, the CWND will be slightly reduced.

An algorithm that takes the principles of TCP Dual and goes even further is TCP Vegas[17] (1994). Vegas has 3 main differences compared to loss-based algorithms. Firstly it uses RTT metrics and CWND to calculate the expected sending rate (using RTT_{min}) and actual sending rate (using latest RTT). Let us assume the CWND has a size of 20 packets with packets having a size of 1000 bytes, the RTT_{min} is 20ms and the latest RTT is 40ms. The expected sending rate is then calculated: $expected = CWND/RTT_{min}$ which results in an expected throughput of 1000 KB/s. The actual throughput can also be calculated by replacing RTT_{min} with latest RTT, this results in an actual throughput of 500 KB/s. The difference of throughput, calculated by $actual - expected$, is 500 KB/s.

Vegas will then also decide the value of two thresholds where both will have a value expressed in KB/s: α and β. Vegas then uses these two thresholds (Figure 12), as

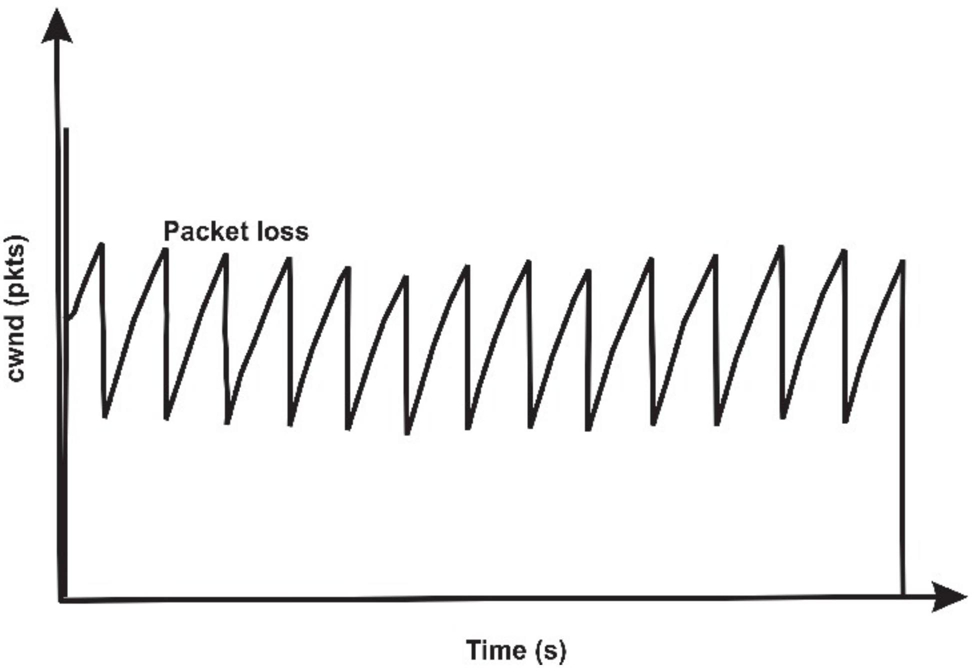


Figure 11: Evolution of CWND in TCP Reno[14]

boundaries to determine how much buffer space per second should be used. Looking at the example described earlier, the CWND will grow if the difference in throughput is lower than α. When the difference in throughput goes over β the CWND will be reduced, otherwise the CWND stays the same if $\alpha < diff < \beta$. The rate the CWND changes will be with one segment every RTT (similar to congestion avoidance in RENO), also for reducing it when congestion is detected. A similar method used as in TCP Dual to reduce the CWND oscillation that is present in loss-based algorithms.

Vegas also changes the way loss is detected in fast recovery, using RTT metrics instead of the three duplicate ack threshold. For each packet acknowledged, Vegas will store the associated RTT value. When the first duplicate ack is received, Vegas will compare the RTT value of the acked packet with the send time of the packet that is potentially lost. If it differs too much from the timeout value, Vegas will retransmit the data. This check will also happen when the first non-duplicate ack is received. This method has an advantage when dealing with large amounts of loss, where less than three packets arrive with a higher sequence number. The receiver will send duplicate acks but the duplicate ack threshold of three acks will not be achieved. So the receiver will then have to wait for a timeout event to detect the lost packets.

The slow start method used in the previous loss-based algorithms can cause a large amount of loss, especially in large Bandwidth Delay Product (BDP) networks. The BDP gives the amount of data that is currently being sent over the network. Due to the CWND doubling every RTT, the longer the algorithms stays in this phase the larger the increase

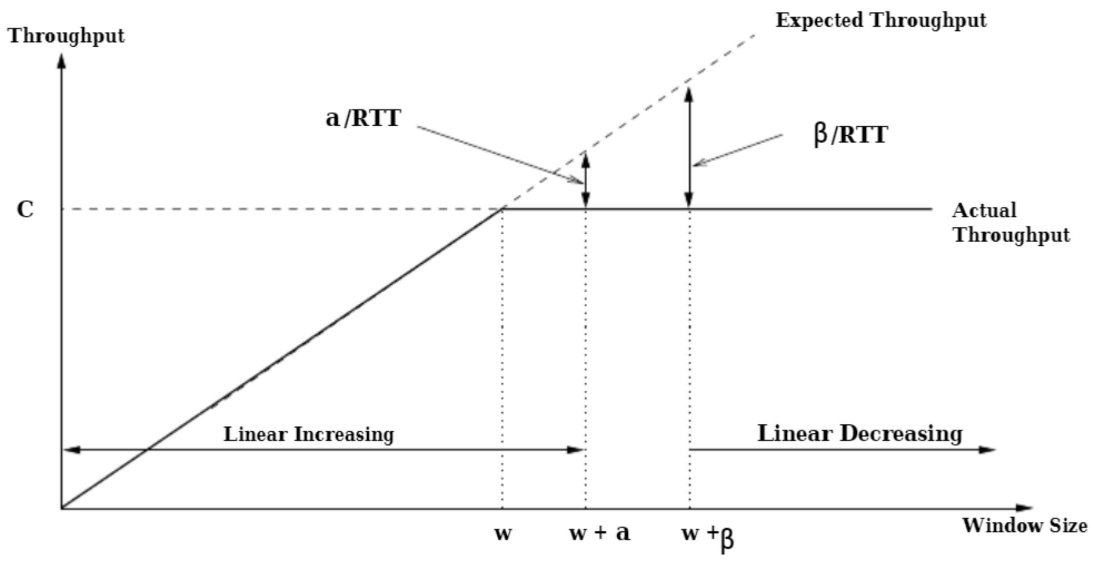


Figure 12: Evolution of CWND in TCP Vegas [16]

will be on the CWND. It can occur that the previous CWND barely makes it through the network without loss, let us assume it fills up the queue by 90%. When the CWND then doubles when the queue only had 10% space left previously, the queue now fills up completely and will drop a large portion of the packets. Vegas modifies the slow start to also include the congestion avoidance method, so it will calculate the difference in the actual and expected throughput again. The CWND will still double in size but only for every other RTT, so that the actual throughput can be measured when some packets of the previous RTT might still be in the queue. Once the difference in throughput goes over the threshold γ, Vegas will switch to the congestion avoidance described earlier.

TCP PERT[18] (2007) changes a couple things compared to Vegas, where the main changes lies in detecting congestion. Rather than using an equation that basically generates a binary result, increase/decrease CWND, PERT generate a back-off probability based on RTT metrics. This done to prevent the CWND getting reduced, for example in Vegas where the difference in throughput barely goes over the β threshold. The slight build up of queue size could have been from the short connection sending a burst of packets, for example to download a short text file. Once the short connection has send all the data, the queue size could lower by itself since the amount of data arriving could be less than what is being sent out by the node.

PERT use an RTT metric different from just using the latest RTT, it uses a more weighted approach where a smoothed RTT is calculated from values of multiple packets, in order to prevent outlier values to affect the CWND. Using the RTT_{min} value two thresholds are defined: Th_{min} which is $RTT_{min} + 5ms$ and Th_{min} which is $RTT_{min} + 10ms$. With the previously mentioned thresholds and the SRTT, the back-off probability can be estimated (Figure 13). It is 0% when SRTT is lower or equal to Th_{min} and will be 5% when it is

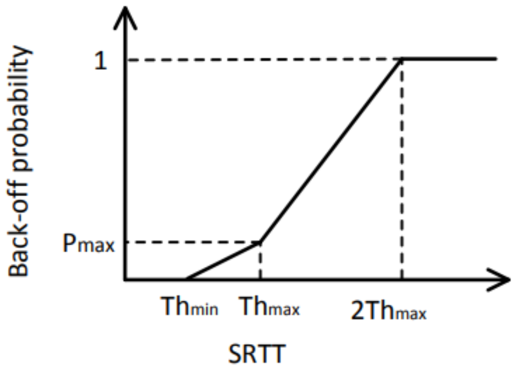


Figure 13: Backoff probability value based on SRTT [15]

equal to Th_{max}. From Th_{max} it can grow to 100% when SRTT reaches $2XTh_{max}$. Using this probability a random number generator can then decide to back-off and reduced the CWND by 35%. Since there is no packet loss yet, it is not necessary to reduce the CWND by half since the queue is not filled up yet. In case packet loss does happen, PERT then reduces the CWND by half.

3.2.4 Competing with other loss-based connections

When delay-based algorithms are competing with each other, they can perform fairly well since all of them try to keep the queuing delay low. However, that changes when loss-based connections are competing that will keep increasing the CWND till the buffers are full. Delay-based algorithms will notice an increase in queuing delay due to loss-based algorithms and will reduce their CWND to try keep the queuing delay low. Thus, loss-based algorithms will take up more resources unfairly from the delay-based algorithms. In Figure 14 we see a Vegas connection achieving a much lower throughput when competing with a Cubic flow. There are a couple algorithms that try to prevent that by having two or more different modes to work in. One of those modes will be an aggressive approach in order to compete with loss-based algorithms.

An early proposal was a modification to Vegas, where the algorithm would switch to Reno if it detected other loss-based algorithms on the network. TCP Vegas+[15] (2000) will first keep track of the amount of times the RTT increased when its CWND was unchanged, it will also decrease this value if the RTT has not increased. If the RTT increases over 8 times when the CWND remained unchanged, which most likely means there is a loss-based algorithm on the network, Vegas+ will change to Reno.

When looking at per packet RTT values, these can differ by a large margin sometimes due to bursts in packets. This can happen because of other connections performing slow-start, where every RTT they double their CWND. Due to spiking values, a delay-based algorithm assumes the network is more congested and will respond to it. TCP CDG[19]

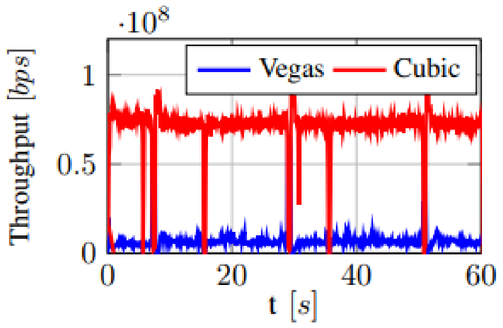


Figure 14: Vegas struggling against Cubic when competing for throughput [12]

(2011) switches to using a delay gradient, it will calculate a smoothed RTT during an interval. This interval has a length of one round trip, which contains all the packets that were sent in short succession of each other. In that interval a minimum and maximum RTT value are used. Whilst using the two measured RTT values and comparing them to the previous interval, the algorithm can also detect if the bottleneck queue is empty, filling up, full or becoming empty. Thus, when loss occurs the algorithm can differentiate if it happened due to congestion or another reason (corruption). So TCP CDG, will only reduce the CWND if loss occurs when the queue is full.

Compound TCP[20] (2005) uses a different method to deal with fairness when competing against other algorithms. Rather than changing the CWND equation depending on the presence of loss-based approaches, it combines a delay-based equation with a loss-based one at all times. CTCP keeps tracks of two different CWND variables: CWND which is the result of Reno, DWND which is the result of the new delay-based equation. The resulting window for the connection is the sum of both values, Figure 15 shows how it evolves during the lifetime of a connection. The delay-based method allows the DWND to grow rapidly when the link is underutilised. By using the sum of both windows, the DWND grows the total window fast when the CWND is slowly growing due to Reno's congestion avoidance. However, like with TCP Vegas the DWND will be reduced slightly when the amount of queued packets goes over a certain threshold. This will stabilise the total window since the CWND is still growing by one packet every RTT, the DWND reduces with a similar rate delaying the total window from growing to large value and causing loss due to congestion.

3.2.5 Advances in loss detection

With packet loss still happening on the internet, possibly caused by using loss-based algorithms, some different approaches to detecting packet loss were introduced. With the primary focus of detecting packet loss early and reducing the amount of spurious loss detection due to acknowledgements being dropped or when packets are being sent via a path with a lower delay causing reordering.

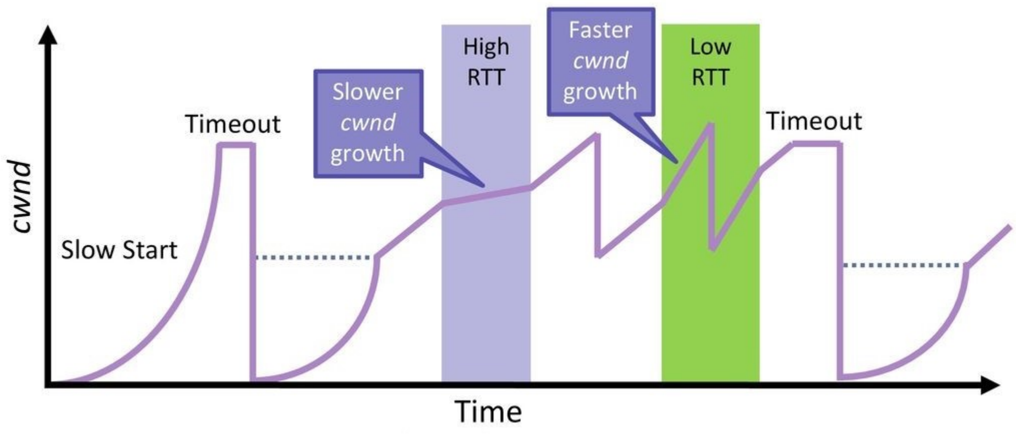


Figure 15: Evolution of congestion window in CTCP SOURCE: `https://slideplayer.com/slide/13451090/`

First we will look at a new timer method to detect loss which is called Tail-Loss Probe (TLP [21], 2012) Before the TLP, the retransmission timeout (RTO) was used. However, researchers have discovered that the RTO would detect more loss than the duplicate ack method. The downside of RTO is that it waits for a long time before marking packets as lost, to prevent it spuriously detecting loss if there is a long delay and acknowledgements arriving after a long round trip. The RTO is expected to trigger if the last packets are dropped and the receiver does not receive any packets to trigger an acknowledgement. It can also happen in the middle of the connection if large amounts of loss occurs and that the receiver does not send more than two duplicate acks. The long wait time for the RTO was delaying the response time for HTTP traffic by a large margin.

TLPs were introduced to help speed up the process by using a shorter timer. However, to not cause spurious loss detection, a TLP timer would not detect packets as lost. It is meant to send some data to the receiver which in turn would trigger an acknowledgement, allowing the ack-based method to detect loss faster than the RTO. At the end of a connection when there is no new data available, it retransmits the last packet. This is for when the last packet sent arrives last at the queue where it has the highest chance to be dropped, in case where the queue size was increasing by the rate of the packets being sent. In the middle of the connection when there is more data available, it would send a packet containing new data. It is possible that all the packets have arrived at the endpoint but the acknowledgements that were sent could have been lost due to congestion in the other direction.

A new ack-based method is Recent Acknowledgement (RACK [22], 2015), which is a more recent loss-detection method that uses time thresholds rather than packet number threshold. The idea for this method came from observations of loss and reordering patterns. Three different patterns that were common have been identified.

First there is the problem of retransmissions being lost again due to traffic policers dropping an excess amount of packets, when the throughput is higher than allowed. It can also happen because of burst losses which causes a burst of retransmissions, which can be lost again if the network is still congested. A second loss pattern is that of Tail loss, where with short living connections (e.g. HTTP requests - responses) loss would happen at the end of the connection where the CWND was the largest. Because the server cannot send anymore data, the packet threshold method would not detect any loss due to no or less than three duplicate acknowledgements arriving.

The third pattern was that of packet reordering, where packets of the same connection could travel over different paths, sending newer packets over another path that has a lower queuing delay. A packet could be delayed long enough that the packet threshold method would detect loss due to other packets with higher sequence numbers arriving earlier.

The existing packet threshold methods do not work well because the way they are configured, means they can detect loss either fast or accurate. A lower threshold means that loss will be detected earlier but would also spuriously detect loss because of reordering. A higher threshold would work better at detecting actual lost packets, but will take more time due to more duplicate acks being needed to mark a packet as lost.

RACK means to solve these issues by switching to a time threshold, using RTT measurements and timestamps to detect loss. For each packet, the send time will be stored and the latest RTT will be calculated for each acknowledgement. Once a duplicate ack is received, with SACK information included, the unacknowledged packets can be checked if they are lost. The time difference between the send time and the time of the acknowledgement arriving for each unacknowledged packet is calculated. If this time difference goes over the threshold, the packet will be marked as lost. The latest RTT is used to take into account a build up in queuing delay for the last arrived packet. If the time difference is not over the threshold, a timer will be set for the remainder as to prevent an RTO from happening. An RTO would cause the congestion controller start from slow-start again.

The RACK time threshold is calculated by the following equation:
$threshold = Latest\_RTT + reo\_wnd$. The reodering window ($reo\_wnd$) is meant to prevent spurious loss detection due to reordering. When no reordering is detected the value of reo_wnd is zero, this allows loss to be detected very early. Once reordering is detected by observing if a packet with a higher sequence number was acknowledged before a packet with a lower sequence number. Then the reordering window is calculated by the following equation: $reo\_wnd = (RTT_{min}/4) * reo\_wnd\_mult$. The reo_wnd_mult parameter is used to increase the reo_wnd if a spurious retransmission happened due to detecting loss too early. Increasing reo_wnd_mult is done when a duplicate SACK is observed, signalling the receiver that a packet arrived that was already received. The value of reo_wnd_mult will be reset to 1 if no spurious retransmissions have happened for sixteen recovery periods in a row.

These new loss detection methods can improve the congestion controllers described earlier. However, in the last four years, new algorithms have appeared which introduce new concepts of congestion control.

3.2.6 Latest algorithms

Later on a distinction was made between two kinds of connections: short bursty flows (mice flows) and long standing flows (elephant flows). Existing long standing flows have the ability to achieve an equal share of the bandwidth, short flows will disturb that. TCP Copa[15] (2018) tackles three main issues that were discovered when using delay-based CCs. Two of them we have mentioned earlier which are fairness issues when competing with loss-based algorithms and RTT values being noisy when looking at each packet individually. The third challenge is that finding the true minimum RTT is difficult when existing connections are fully utilising the available bandwidth, which causes queuing delay to be included in the measured minimum RTT.

For COPA to manage its CWND, it first gathers the necessary RTT metrics. First there is the RTT_{min} to have the minimum transmission delay. With the delay being noisy, COPA works with a new RTT variable called $RTTstanding$. $RTTstanding$ is the smallest delay observed within a time window of $SRTT/2$. The reason for taking a minimum delay is better deal with delay noise caused by ack compression. Ack compression can happen when acknowledgements sent by the receiver arrive at a node where queuing is happening, these acks have been spread over a period of time due to data packets experiencing different queue times. Those acks on the queue can be send in quick succession and arrive at the sender at the same time. The resulting RTTs measured for each ack will not be equal, since the first acks arriving might had to wait longer in the queue than the later packets if the queue was being drained faster than it was getting filled. Using a minimum RTT value lets COPA get a more accurate queuing delay. The queuing delay is calculated using an earlier described method: $d_q = RTTstanding - RTT_{min}$.

COPA can now calculate what the ideal sending rate would be by using the following equation: $\lambda_t = \frac{1}{\delta . d_q}$. When the queuing delay is low, meaning the queue is barely filled, the target rate can be high. When the queuing delay starts increasing, the send rate would go down allowing the queuing delay to go down again. The other variable δ is used to adjust the target rate. The default value for δ is 0.5, this reduces the impact of the queuing delay for a bit to allow some packets to be queued. By doing this, COPA can generate a throughput that stays at the maximum bandwidth. This parameter is decreased when other loss-based congestion controllers are detected. COPA can detect these congestion controllers when it sees that the RTT_{min} does not change.

Since COPA uses RTT metrics from a previous RTT window, it will reduce its actual send rate for an additional RTT allowing the queues drain. The resulting target rate will oscillate close to the ideal value, which we can see in Figure 16. The actual send rate will be calculated by the following equation: $\lambda = CWND/RTTstanding$. If actual send

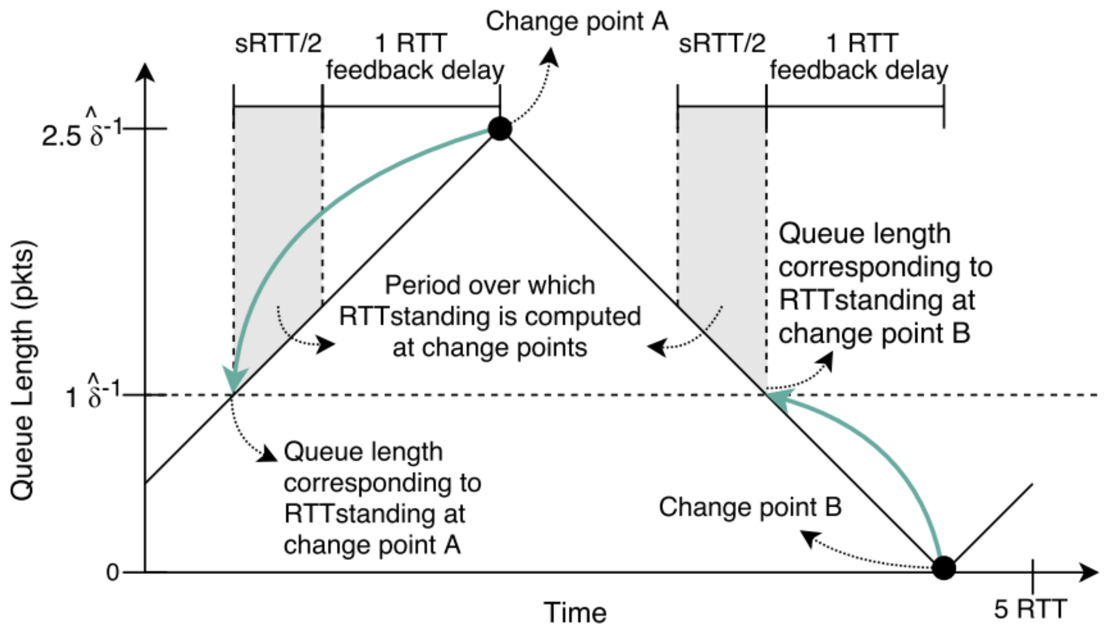

Change point A
sRTT/2
1 RTT feedback delay
sRTT/2
1 RTT feedback delay
$2.5\ \hat{\delta}^{-1}$
Queue Length (pkts)
Period over which RTTstanding is computed at change points
Queue length corresponding to RTTstanding at change point B
$1\ \hat{\delta}^{-1}$
Queue length corresponding to RTTstanding at change point A
Change point B
0
Time
5 RTT

Figure 16: Evolution of queue length within one Copa cycle [23]

rate is over the target rate, the CWND will be reduced: $cwnd = cwnd - v/(\delta.cwnd)$. The velocity parameter (v) can change in value if with the new CWND, the actual send rate is still too high. Only after waiting for three RTTs since the actual rate went over the target rate, can v start to double. This is to check that if the queuing delay is still too high, to not keep such a high transfer rate. When the actual sending rate is lower than the target rate, the same equation is used only that v will be positive. Slow-start for COPA follows the same principle as in Reno, where the CWND doubles in size every RTT until the actual send rate goes over the target rate.

COPA also uses packet pacing to reduce the build up queuing delay by introducing a time gap between each packet sent. Figure 17 shows us what happens when no pacing is performed, all the packets are send as a burst. The resulting throughput of the burst is much higher than the available bandwidth for the router. This causes a build in the queue, introducing an extra delay as well as leaving less room for other connections. By introducing a time gap for each packet send, the resulting throughput is much lower. This allows the router to send packets out at around the same rate as they are received, reducing the amount of queued packets. This also stabilises the measured RTT for each packet, in the example with no pacing the RTT will increase for each packet since it needs to stay in the queue longer.

There is another algorithm that was developed by google, named Bottleneck Bandwidth and Round-trip propagation time or BBR[13] (2016). TCP BBR was designed operate at the optimal point which is described in section 3.2.3. The main method to BBR is to measure the maximum bandwidth and the minimum RTT, however these would be

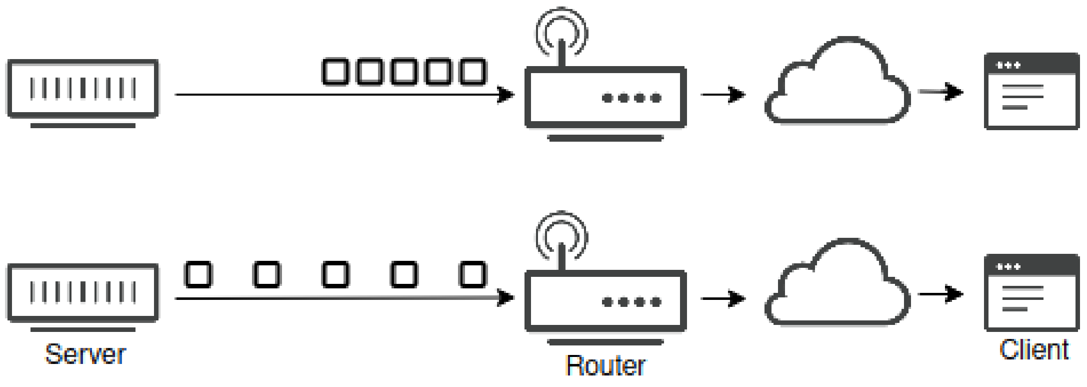


Figure 17: Build up in queue without pacing and with pacing source `https://blog.cloudflare.com/accelerating-udp-packet-transmission-for-quic/`

measured at different times because they influence each other. When measuring the minimum RTT, packets cannot be queued. In order to prevent packets from getting queued, the TCP endpoint needs to send data at a slower rate than the network can handle. However, this slower rate does not equal to the maximum bandwidth. To find out the the maximum bandwidth, more data is send to detect a build up in queuing delay. This does increase the measured RTT, since queuing delay is also included in the measured RTT.

With these two values the BDP can be estimated, which will be used as the ideal CWND. BBR finds the minimum propagation delay (RTT_p) by observing the minimum RTT. The available bandwidth (BW_{btl}) can be observed by processing the acknowledgements, by looking at how much data is acknowledged in a time interval. BBR uses the same slow-start phase where the CWND is doubled for every round trip. Rather than waiting for packet loss to indicate congestion, BBR checks if the calculated throughput has not changed for three round trips. The threshold of three round trips is used, to prevent a burst of packets to influence BBR. The burst of packets can stay in the queue long enough for BBR to detect the throughput has not changed. The burst of packets can be processed fast enough, allowing for more bandwidth to be used.

By using the threshold a build up in the queue can happen. So when exiting slow start, BBR switches to the drain phase. In this phase the CWND is reduced to 75% of the calculated BDP. After draining the queues, BBR switches to PROBE_BW where it will probe for more bandwidth. It does by increasing the CWND to be 125% of the calculated BDP. If the calculated bandwidth (via acks) does not change, that means the available bandwidth hasn't changed. Because of this a build up in queuing happened and BBR switches to the drain phase again before continuing with the original CWND value. Otherwise BBR does not drain the connection and uses the new CWND.

If there has not been a change in the minimum RTT for a few seconds, BBR switches to

PROBE_RTT to find a new minimum RTT. It does this by reducing the CWND to four packets for one round trip. This prevents a build up in queuing, whilst also filtering out packets that may have experienced queuing due to other connections. It could happen that data is being sent over another path, where the propagation delay has changed.

Since BBR doesn't use loss as a signal for congestion, it can maintain a high throughput in shallow queues where occasional loss happens. However, this causes fairness issues with loss-based approaches. If a small queue is present in the network, this can cause issues for loss-based approaches. Not long after the CWND has grown to the point the throughput matches the available bandwidth, the CWND grows too large where the small queue is filled up. The CWND then is reduced since it detected congestion, but the resulting CWND now results in a throughput that is lower than the available bandwidth. BBR amplifies this problem due how it operates around the optimal point. It periodically increases its send rate to probe for more bandwidth, when there is a small queue, this increase can fill it up very quickly. A loss-based approach that is also on the same path, can experience loss faster and reduce its CWND. Currently a new version of BBR is being developed to fix this issue and improve the algorithms in other scenarios[4].

3.3 QUIC's choice

We have showcased several different congestion control algorithms which could be used for the new QUIC protocol. However, not every algorithm is a good choice and it will also depend in what environment QUIC will be deployed. An example is a data center environment where high bandwidth is available and the propagation delay is low, an algorithm that achieves the same goals as DCTCP would be optimal. We can still look at the different congestion control concepts that are being used and analyse which would make a good option.

When looking at the different loss-based approaches and comparing them to delay-based ones, we can see that the former approach causes problems for itself and for the delay-based approaches. In the current environment of the internet, filling up buffers causes extra delay on all packets as well as leaving little to no room for new connections. Delay-based approaches are having difficulties with achieving a high throughput when competing with loss-based approaches. When looking at the latest algorithms we see large differences with all the earlier approaches. With the internet itself becoming more and more complex, so did the congestion control algorithms. It's not an easy choice for QUIC to find or create a good congestion control algorithm.

The IETF working group for QUIC has created a loss detection and recovery document[24] detailing a new congestion control algorithm based on NewReno. It also explains the different loss mechanisms that are used which were not in the original NewReno algorithm. However, the choice to base the algorithm on NewReno is an interesting one.

[4] `https://www.ietf.org/proceedings/104/slides/slides-104-iccrg-an-update-on-bbr-00`

We have discussed that loss-based approaches, such as NewReno, are not the most efficient anymore. A reason for choosing NewReno was because it was one of the only algorithms that is an IETF standard. The working group decided that designing a whole new congestion control algorithms was out of scope[5]. The choice for using a congestion controller that is standardised, was to allow Cubic to be chosen if it was standardised before QUIC[6]. For loss recovery there are more options available, compared to when NewReno was initially used. This new congestion control algorithm with its loss detection methods is written as an example, developers are allowed to use other algorithms. Over time numerous changes have been made to QUIC's NewReno algorithm, which we will look at further in this thesis to evaluate the impact of these changes. First we will describe how we can evaluate the new congestion control and loss recovery methods.

[5] `https://datatracker.ietf.org/wg/quic/about/`

[6] `https://mailarchive.ietf.org/arch/msg/quic/eJBlfPzJ_s4MYcCAwup6MJYAN1o/`

4 Testing setup

In order to analyse the differences in QUIC congestion control and loss recovery compared to TCP, we have to perform a wide range of tests. For this we need the following components: implementations, configured networks and metrics to compare. We will discuss each component in greater detail in this section.

4.1 Implementations

There are currently 18 different QUIC implementations available that are being updated regularly. However, not every implementations supports the QUIC recovery draft, which limits our options. We have decided to use two different implementations which were slightly modified to allow extensive testing: Quant[7] and Aioquic[8].

Quant is a C implementation focusing on system level performance with using a zero-copy method when sending and receiving packet. It also followed the recovery standard on all levels, compared to Aioquic. Aioquic is a Python implementation where high system performance is less likely due to the language limits. The congestion control was modified when compared to the recovery standard. With these very different implementations, we could also look for performance issues in Aioquic due to the Python language. It will also be possible to see which modifications were done to Aioquic compared to the recovery draft and how these would impact the performance.

With the goal being to compare QUIC congestion control and loss recovery to TCP, we will also need a TCP implementation to evaluate the QUIC implementations. With TCP being implemented in the OS kernel, this limits our options. We have chosen to use Linux as our kernel, specifically version 5.4 which is a more recent version. Using such a recent version is important since TCP loss recovery has changed over time. There is another reason we have chosen for Linux but will be discussed in Section 4.3.

4.2 Network simulation

In order to run different kinds of test for different network configurations and behaviours, we use two different network simulators. There are public endpoints available for different QUIC implementations, however the state of the network is volatile and cannot be managed. Currently there is a framework available which includes different network behaviours as scenarios, namely Quic-Network-Simulator[9].

[7] `https://github.com/moonfalir/quant`
[8] `https://github.com/moonfalir/aioquic`
[9] `https://github.com/marten-seemann/quic-network-simulator`

QNS uses the NS-3 simulator[10] to simulate different types of networks and allows QUIC implementations to connect via the use of Docker containers (Figure 18). It is also possible to connect containers that send out data via TCP. QNS contains different scenarios like introducing packet loss, network shutting off, fairness test against a TCP/UDP flow, ... Network parameters can be supplied to configure the link between client and server: bandwidth, delay, queue sizes, ... With some minor adjustments we are able to use this to run different tests with QUIC and TCP.

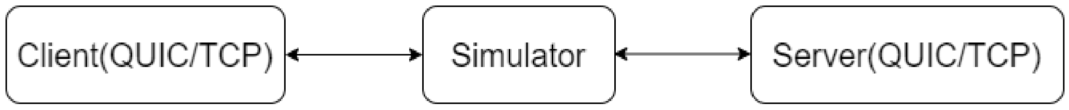


Figure 18: Setup of docker containers and flow of data

4.2.1 Mininet

However, to improve the quality of the test results we want to use a second framework based on a different network simulator. With a second simulator we can try to separate specific behaviour, to be either caused by the implementations or by the simulator itself. For this we have chosen to work with a modified version of Containernet[11], which is a fork of the Mininet network simulator[12].

Mininet uses Linux based network emulation TC Netem and other network virtualisations such as OpenVSwitch to create network topologies. The containernet fork adds the support of docker containers as hosts, which allows us to recreate the same structure as QNS to perform tests. We also recreated the different scenarios that are available in QNS, the ones we used will be explained in Section 5.

4.3 Test framework

To make it easier to run tests for the different implementations and on the different network simulators, we created a test framework[13] that automates most of the work. Figure 19 shows the basic steps which the test framework goes through, this is repeated for each combination of client/server, test configuration and type of simulator (NS-3 or Mininet). With the availability of VMs, we could run these test in parallel. The test framework uses a list of implementations and a list of test scenarios to divide up the tests to each VM. Each server will run the series of tests and signal that it has finished.

In order to analyse the results we also need to gather data about the implementations. We gather pcap files to get information about packets being sent on the wire. For the

[10] `https://www.nsnam.org/`
[11] `https://github.com/moonfalir/containernet`
[12] `https://github.com/mininet/mininet`
[13] `https://github.com/moonfalir/quicSim-docker`

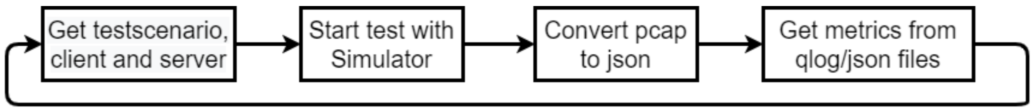

Get testscenario, client and server
Start test with Simulator
Convert pcap to json
Get metrics from qlog/json files

Figure 19: Steps the test framework undertakes to perform tests and gather data

QUIC implementations we also need extract the SSL keys to decrypt the packets, this does not need to be done for TCP. Most QUIC implementations also have the option to output qlog[14] files. This is a logging schema defined for the QUIC protocol to allow clients/servers log packets send/received but also much more like congestion control, loss detection, internal errors, ... This provides a lot of usable data to analyse QUIC behaviour.

4.3.1 Extracting CC information from TCP

Sadly TCP does not have a logging schema implemented in Linux and the available data in pcap files does not provide much insight in congestion control and loss recovery. With the information being available in the kernel, we need a method to extract it. Linux has that option available with the use of eBPF (extended Berkeley Packet Filters). This method allows user code to be compiled and run in the kernel itself. With this, it is possible to write functions that can access TCP congestion control and loss recovery information. Using BCC[15] we have implemented trace functions that are bound to TCP kernel functions. These self implemented functions extract similar information that would be present in qlog. The acquired data is sent up to user space to the python front-end, which gathers the events and converts the data into qlog format.

However, creating the tool which includes the C code was more difficult than expected. An example would be to trace the CWND and log each time this increases, for slow start and congestion avoidance. In Linux one function is used, called `tcp_reno_cong_avoid`, to increase the CWND. Even thought it has the name `cong_avoid`, it will also perform the slow start method.

Using a Kprobe we can bind our trace function to this `tcp_reno_cong_avoid` function, which can extract the CWND value and send it up to the python front-end. The trace function has the same parameters as the kernel function which it is bound to. When the kernel function is called, the Kprobe will first call our trace function first before it continues with the kernel function. However, this causes an issue since the CWND has not been updated yet when our trace function is running. We fixed this by linking the CWND data to the timestamp of the previous event, so the event at t0 has the data which was gathered at t1. Using this method we are able to get the values after the update has happened, also for other values than the CWND.

[14] `https://github.com/quiclog/internet-drafts`
[15] `https://github.com/iovisor/bcc`

Another difficulty was finding the right kernel function to bind a trace function to. Some kernel functions were `inline` and were not known to the compiler used for the trace functions. In this scenario, we traced back the kernel function call and see which usable kernel function would call this `inline` function.

In the end it was possible to retrieve the same data that is present in the QUIC qlogs, which allowed us to also use qvis for TCP and compare it to the qlogs of the QUIC implementations.

4.4 Analysing results

We also added a metric calculator to the test framework, which parses the previously mentioned files to gather metrics. There is also a visualisation tool available for qlog and pcap files named qvis[16], which allows us to easily analyse test results in greater detail. Figure 20 shows a qlog file loaded into qvis, this show us information of the entire connection. For example, we can observe how the CWND (purple line) evolves over time.

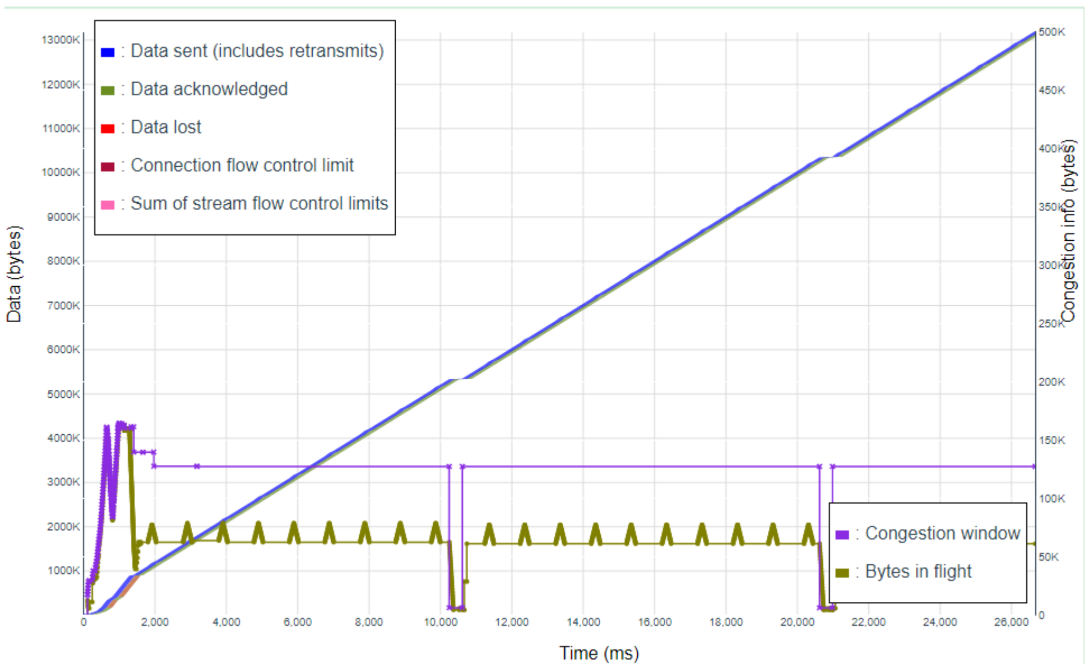


Figure 20: Example of qlog file in qvis, showing packet and congestion control information

To analyse the performance of TCP and QUIC we keep track of multiple metrics that are related to congestion control and loss recovery. We will list the metrics that we use and explain why we are using them.

[16] `https://qvis.edm.uhasselt.be`

- Average Throughput (KB/s): We keep track of the average throughput to see how close the implementation gets to the maximum available bandwidth.
- Average Goodput (KB/s): The goodput represents the rate of actual data being sent, data is represented in the payload. We remove the overhead from protocol headers, QUIC frames that do not contain data and retransmissions. This can give us an indication of how efficient the implementation is with regards to sending data.
- Average Latest RTT: We also want keep track of the average RTT experienced for the packets. This can give an indication how much longer the measured delay is compared to the configured delay.
- Retransmissions: We will also look at the total amount of retransmissions during the lifetime of a connection. This give us a clear picture on much loss is detected and is also important for the following metric.
- Spurious retransmissions: Newer loss detection methods were introduced for TCP that should reduce the amount of spurious retransmissions. The amount of spurious retransmissions can have a large impact on the performance of NewReno, by tracking this value we can identify any performance issues that is related to spurious retransmissions.
- Triggers for loss detection methods: In Section 3 we highlighted a few different methods used to detect loss, which can be used at the same time. To get a better understanding on how loss is detected in TCP and QUIC, we track for each lost packet how it was detected as lost. This can also help when spurious retransmissions are happening, since sometimes packets are marked lost unnecessarily.

One difficult metric for QUIC to detect is a retransmission, which is not as simple compared to TCP where packet numbers are re-used. As described in Section 2.2, QUIC has moved the offset for the data into the payload. It is that offset that does get re-used if a transmission happens. So in order to detect a retransmission we have inspect the stream frame for the offset and see if it was already seen previously. To detect a spurious retransmission we have to go back to packet numbers, where we keep a list for each stream offset, which packet numbers were used. If we see that a stream offset has been acknowledged in a previous packet, we can track it as a spurious retransmission.

4.5 Simulation bandwidth and queue

One of the key parts of a network simulator is to simulate a bandwidth. First, what is the bandwidth and how is the data sent according to the bandwidth. The bandwidth is a rate of data being send over time, so how much data can be sent over a period of second for example. When looking at how data is sent over the wire or the air, it is done by sending a sequence of bits. Depending on how many bits need to be sent, it can take some time to send an entire packet. We will be looking at how data is send over a wired connection, since with WiFi data is send via a different method.

When a packet arrives at a node it needs to be inspected to know where to route the packet to, this takes some time although not much. Once it has been determined where the packet needs to go, it can be directed to the correct network interface. There it can be send over the wire, which takes some time. The rate at which packets are being sent is a constant rate and between each packet there will be a time gap (Figure 21). So for a network simulator this is important to simulate where a constant stream of packets is virtually send to the next node.

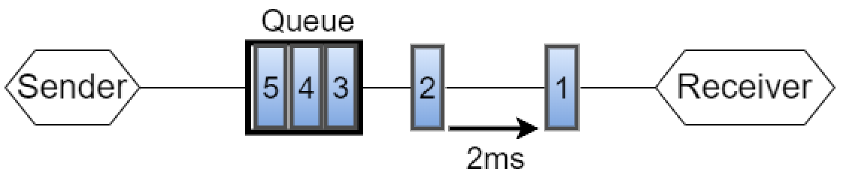


Figure 21: Example of bandwidth simulation: packets are send over the wire where there is a slight delay between each packet

Before inspecting a packet, the packet will first need to be removed from the queue. The queue is there to catch a burst of packets arriving, which are not being processed at the same time. Using a queue gives the node enough time to be able to process and send each packet off one by one. How often a packet is removed from the queue depends on the bandwidth, where each time a packet is sent and new one will be popped from the queue. This means that with a simulator packets need to be removed from the queue at the same rate as the bandwidth.

In order to make sure that the simulators behave as we have just described, we will have perform some tests. We can evaluate these test with the help of qvis and logs from both client and server.

4.6 Comparing simulators

The goal of using two simulators is to help differentiate behaviour of a simulator from the QUIC or TCP implementation. The main components of the simulators that could differ is the method of bandwidth simulation and packets queuing/dequeuing. To evaluate the two simulators we have, we performed some initial testing. We used the two QUIC implementations and Linux TCP to help with identifying which behaviour comes from a simulator.

For the first test we used a configured delay of 5ms and a bandwidth of 5Mbps. For queue size we determined the size by calculating the BDP: $BDP = delay * BW = 5ms * 5Mbps = 3125Bytes$. Reason for this small queue size is to also emulate the presence of other connections, to see if the implementations can optimally use the available bandwidth with limited queue size. With the queue size needed to set in amount of packets, we divided the BDP by the Maximum Segment Size. We chose for a MSS of 1280 bytes, which is the minimum size the QUIC packet can have[1]. By using the MSS we can calculate a queue size of: $Q = 3125/1280 = 2.44$. We rounded this value up to three to allow for one more packet to be queued, since a value of two packets is quite small (packets might not be removed from queue fast enough).

Using the test framework we ran this test for each implementation five times, to see if using the same configuration would give similar results. We do not expect deterministic results but large differences are not welcome. The goal also is not to focus on the differences in performance between two implementations, this will be done in Section 5.

Implementation	Throughput(Kbps)	AverageRTT (ms)	#Retransmissions
NS-3			
Aioquic	4579.21	16.93	153
Quant	4608.39	17.64	76
Linux(TCP)	4836.3	16.31	113
Mininet			
Aioquic	2188.56	11.86	1378
Quant	2482.99	12.07	386
Linux(TCP)	2310.95	10.81	152
Implementation	Throughput(Kbps)	AverageRTT (ms)	#Retransmissions

Table 1: Initial test results with 5ms delay, 5Mbps bandwidth and queuze size of 3 packets

The results are shown in Table 1, comparing the two simulators shows large differences. We notice that in NS-3 the throughput is larger than in Mininet, close to the 5Mbps limit. In Mininet we see a large increase in retransmissions, indicating that more packets are lost. Using NewReno as a congestion controller would explain why the throughput in Mininet is lower, since it sees loss as a signal for congestion. The question then becomes why there is so much packet loss happening.

When looking at possible causes of packets loss, there is the possibility of packet corruption. However, packet corruption happens due to noise interfering with the transmission over wire or in the air. In a simulator the packet never leaves the server and thus packet corruption could not happen. Another possible cause for packet loss, is due to congestion where the queue is overflowing and packets are being dropped upon arrival.

When comparing the RTT values of both simulator we also see a difference, in Mininet the average RTT is very close to the configured RTT (2 * 5ms = 10ms). If the measured delay goes over this value, we expect it is due to queuing. In Mininet the extra delay is very small, which indicates a short queuing delay. Though the amount of loss would indicate that the queue fills up very fast. It could be that packets that arrive and are configured to experience a delay (5ms), are not immediately removed from the queue, even though the bandwidth limit would allow for packets to be removed.

Given this situation we decided to further experiment where we would perform tests with two different bandwidth values, whilst using a very small queue of two packets. The idea being we would observe an increase in throughput, due to the higher bandwidth allowing packets to be removed from the queue a little faster. We used a bandwidth of 5Mbps and 20Mbps, so that an increase of throughput would be obvious to see. We do not expect a large increase in throughput due to the small queue, where a burst of packets can easily overflow it. A delay of 10ms is used to make sure that if packets are kept in the queue due to the delay, they stay the for a significant time.

Implementation	Throughput(Kbps)	AverageRTT (ms)	#Retransmissions
5ms, 5Mbps, queue 2 packets			
Aioquic	4169.53	25.5	95
Quant	4157.42	26.83	60
5ms, 20Mbps, queue 2 packets			
Aioquic	5487.26	22.72	78
Quant	3316.74	22.86	112
Implementation	Throughput(Kbps)	AverageRTT (ms)	#Retransmissions

Table 2: Test results of queue size 2 (packets) with two different bandwidths using NS-3

We see the results of the tests for NS-3 in Table 2 for both bandwidth values. Looking at Aioquic we do see behaviour that we expected: increasing the BW did result in an increased throughput, it also is a small increase which is due to the small queue size as explained earlier. The RTT is lower in the 20Mbps bw scenario: this is the result of simulating the bandwidth where the gap between each packet is larger when the limit is 5 Mbps, shown in Figure 22.

When looking at Quant we see a different result when the bandwidth increases, we see the throughput decrease. Looking at the qlogs in qvis using the congestion view: we see that with 5 Mbps the CWND grows and reduces following a consistent pattern. Comparing that to the scenario with 20 Mbps bandwidth, Quant goes into recovery in quick succession at some times. We attribute it to Quant growing the CWND faster in

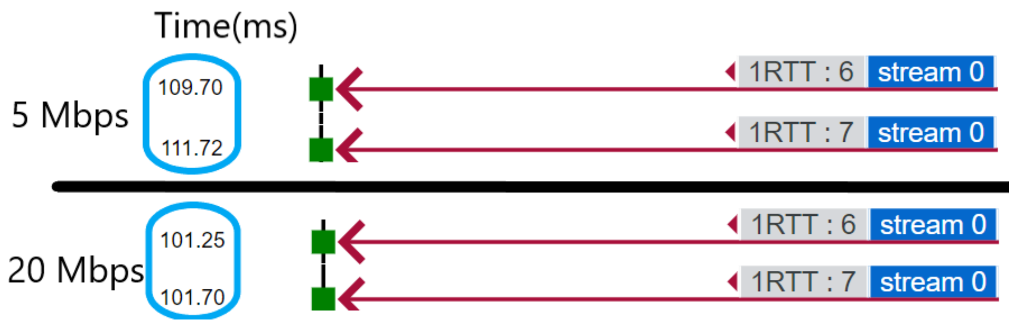


Figure 22: Time difference between two packets arriving with two different bandwidths in NS-3

20 Mbps, due to acks arriving faster (ack compression). After recovering, Quant sends too many packets in a short burts due to acks arriving at a faster rate, causing some packets to be lost. Quant enters recovery and reduces its CWND by half again, not long after it did it the first time resulting in a CWND that is $\frac{1}{4}$th the value.

So in NS-3 we see what we would expect for Aioquic, a slight increase in throughput when the bandwidth is increased. Let us repeat the tests with the same configuration, but this time in Mininet. If we do not see a difference in throughput, it might be that the queue is limiting factor.

Implementation	Throughput(Kbps)	AverageRTT (ms)	#Retransmissions
5ms, 5Mbps, queue 2 packets			
Aioquic	1068.17	22.25	904
Quant	1066.82	22.04	620
5ms, 20Mbps, queue 2 packets			
Aioquic	1070.72	22.18	922
Quant	1059.92	22.13	621
Implementation	Throughput(Kbps)	AverageRTT (ms)	#Retransmissions

Table 3: Test results of queue size 2 (packets) with two different bandwidths using Mininet

First when we look at the result we do not see any difference in throughput, so increasing the bandwidth had no effect. Also looking at the average RTT and amount of retransmissions, show little to no difference as well. We have the option available to look at each test in detail using qvis, to see what is causing the high amount of loss. We expect that most loss happens due to the queue overflowing, so something with the queue could be happening in Mininet which does not give the expected results. We will first look at the sequence diagram of Aioquic for the NS-3 simulation, since it seems that NS-3 is behaving as expected.

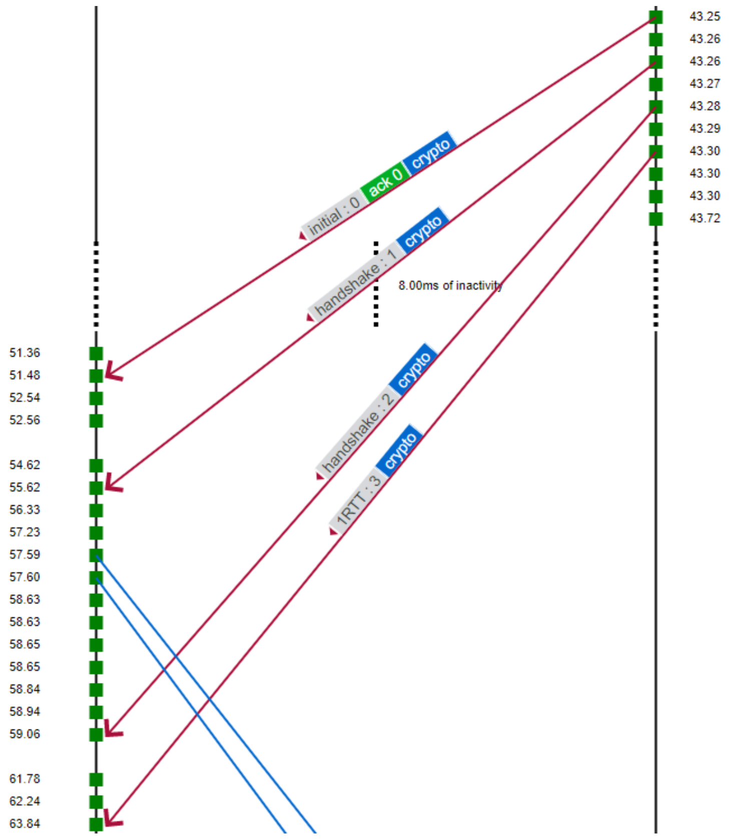


Figure 23: Handshake performed by Aioquic server in NS-3 with queue of 2 packets, 5ms delay and 5Mbps BW

We can see in Figure 23 how the handshake is performed between the Aioquic server and client, where we see that all four packets (red arrows) arrive at the client. Event with a small queue size of two packets, the first two packets are removed from the queue fast enough so the other two packets can be stored. We will now look at the sequence diagram of Aioquic for the Mininet simulation. If we see that that not all handshake packets are received at the client, Mininet might have a problem with removing packets from the queue fast enough.

In Figure 24 we can see something different happening, the last two handshake packets never made it to the client. This behaviour is seen throughout the connection, where only two packets are received by the client if a burst of packets is sent in a short time. When we looked at the initial tests we did, where a queue size of three packets was used, we could see the same happening. Only with a queue size of three packets, we could see that now three packets were being received at the time. Due to high amount of loss, the server spends so much time on retransmitting data, that it cannot send much new data after it.

We expect that Mininet does not clear the queue if packets can be sent, even when the

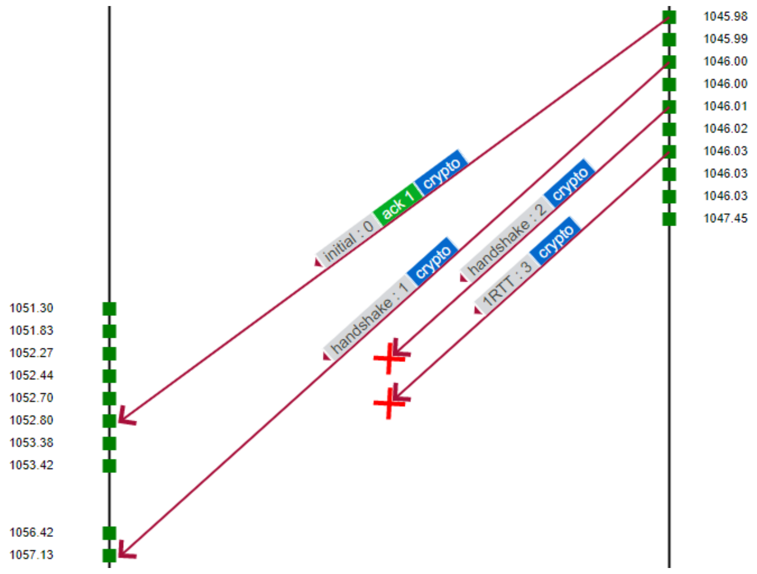


Figure 24: Handshake performed by Aioquic server in Mininet with queue of 2 packets, 5ms delay and 5Mbps BW

configured bandwidth should allow packets to be removed from the queue at a steady rate. In Figure 25 we can see an example of how the configured (propagation) delay is emulated. In NS-3 we see that packets are being removed from the queue and then experience the delay, which is what happens in a real network. However, when looking at Mininet, the packets experience this delay whilst being in the queue. When the packets are being removed from the queue, they immediately arrive at the receiver without delay. This problem has been experienced before very recently[25], where researchers discovered that Mininet's queue limit actually limits the amount of packets that can be in-flight. It is caused by how Mininet configures the queue size, Mininet uses `TC Netem` to configure both the delay and queue size. However, this queue is used for delayed packets to stay until the timer expired, thus works not as a queue compared to NS-3.

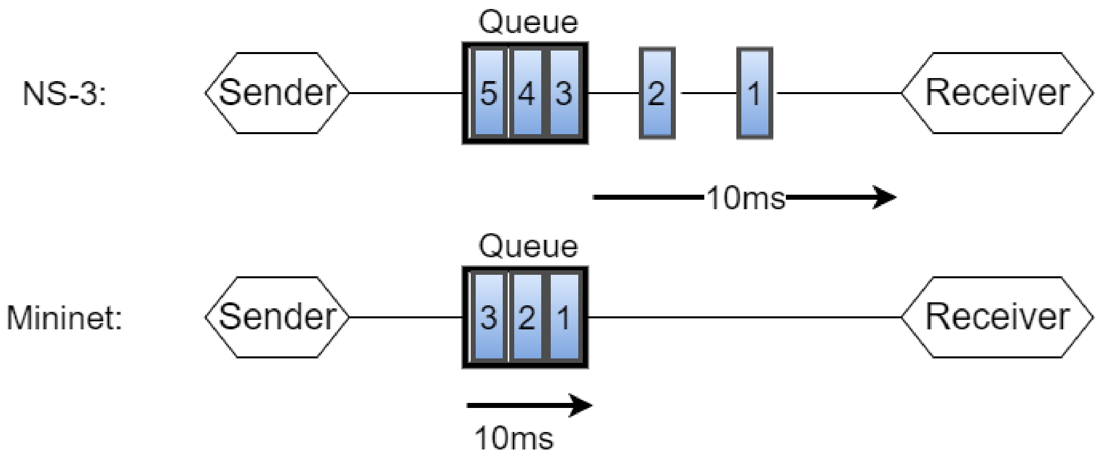


Figure 25: Propagation delay emulation in simulators

In order to make sure the problem lies in the queuing but also to see if we can rectify the problem Mininet has, we decided to perform some tests where the queue size is twice the BDP. If this simple fix would generate results that are similar to NS-3 which uses a queue size of the BDP, we could still compare the results of both simulators. For this we ran a test with a slightly higher delay (10ms), to make sure this would work when using a higher delay where packets in Mininet will stay longer in the queue. We still used a bandwidth of 5 Mbps and determined the queue sizes: four packets (Q4) = BDP and eight packets (Q8) = 2 * BDP.

Implementation	Throughput (Kbps) Q4	Throughput (Kbps) Q8	#Retrans Q4	#Retrans Q8
NS-3				
Aioquic	4487.14	4795.19	78	52
Quant	4449.49	4833.85	48	46
TCP	4424.52	4786.13	83	61
Mininet				
Aioquic	1636.74	3740.27	819	136
Quant	1731.08	4039.01	471	91
TCP	1608.5	4282.49	218	89
Implementation	Throughput (Kbps) Q4	Throughput (Kbps) Q8	#Retrans Q4	#Retrans Q8

Table 4: Test results of queue size 4 (Q4) and 8 (Q8) packets

Looking at the results in Table 4 we can see that using a larger queue has an effect on throughput in NS-3. More queuing allows for more packets to be received in a burst, which largely happens in slow start, where the CWND grows quite rapidly. We also see that the amount of retransmissions is reduced, which confirms it.

However, looking at Mininet, the throughput has more than doubled for all three implementations. We can also see a great reduction in retransmissions, a larger difference but is explained by the previous paragraph. The resulting throughputs in Mininet when using a queue size of eight packets, are close to the results in NS-3 when using a queue size of four packets. However, the difference is still too large and we see that in Mininet more loss is experienced. This means that simply increasing the queue size to two time the BDP, is not enough.

We decided to try one final time this time using a higher bandwidth of 20 Mbps, where the BDP is larger and would allow for a larger queue. An issue could be that using such small values for a queue, are not ideal for the Mininet simulator where packets need to be removed very fast to make space available. For Mininet we decided to increase the queue to four times the BDP, to see if we would get results that are more inline to NS-3 which still uses a queue size of one time the BDP. The delay has been kept the same at 10ms.

Implementation	Throughput(Kbps)	RTT(ms)
NS-3: Queue = 16 packets		
Aioquic	12542.43	61.77
Quant	12831.79	58.65
TCP	17879.95	23.97
Mininet: Queue = 64 packets		
Aioquic	15457.04	42.16
Quant	15019.32	46.02
TCP	18896.89	32.72
Implementation	Throughput(Kbps)	RTT(ms)

Table 5: Test results of using increased queue in Mininet

Table 5 shows us the results of the tests with using an increased queue size for Mininet. When comparing the two TCP results, we see that in Mininet it achieved a higher throughput than in NS-3. Related to that we also see that the average RTT is higher. This is the result of the increased queue size, where more packets were able to be queued in Mininet. Because of this increased queue size, it could handle larger bursts of packets during slow start. However, with a longer queue comes longer delay, so packets had to spend more time in the queue compared to NS-3.

However when looking at the results for the QUIC implementation we get some conflicting results. At first glance we see the same happening compared to TCP, only that in Mininet the average delay is much lower. Looking at the results of TCP, we expected higher delays due to the increased queue size. However, it looks like packets are being delayed longer in the queue with NS-3.

We also see a large difference in throughput when comparing the QUIC implementations to TCP, whilst also experiencing much longer delays. We will first focus on the difference in RTT values, since with NewReno RTT fairness is an issue. It could be that the increased delays are slowing the growth of the CWND, which would have an impact on the throughput. Firstly we will plot the latest RTTs experienced in NS-3 onto a boxplot, to get a better understanding if the RTT values are spread out or concentrated at a higher value.

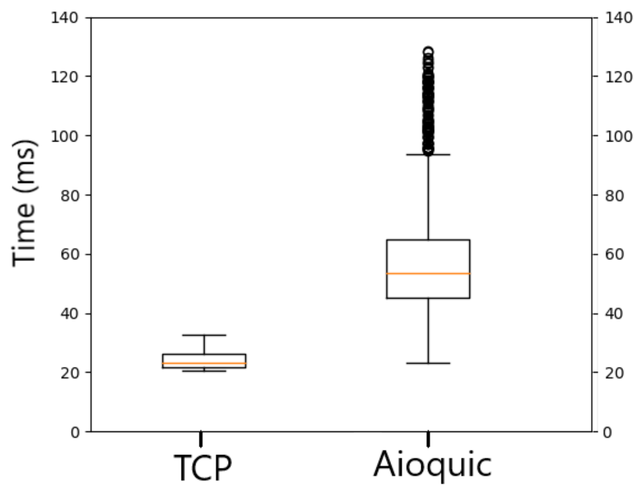


Figure 26: Boxplot of latest RTTs of TCP and Aioquic in NS-3

When looking at Figure 26 we can see a clear difference in terms of range for the latest RTTs in NS-3. With TCP they are much more concentrated compared to Aioquic. With Aioquic we also see a large amount of outlier values (circles). It could be that NS-3 is a bit slower with removing QUIC packets from the queue, so the time gap between each packet is larger. This would also increase the queuing delay by quite a large amount, which then increases the measured RTT. However, when looking at the qlogs we did not see an increased delay between each packet for Aioquic when compared to TCP. We looked at an interval of ten packets for each implementation, looked how long it took to send the ten packets. For both implementations it took around the same time, no obvious and large difference.

What we could see is that for Aioquic the total transfer time was around two seconds longer than with TCP, the increased RTT values do have an effect on it. With it taking more time to receive acknowledgements, it takes longer for the CWND to grow, which keeps the throughput lower for a longer time. We did not find the cause of these increased delays for the QUIC implementations and decided to continue with comparing the congestion controllers whilst taking in these results in mind.

Another small issue we found had to do with calculating the goodput and (spurious) retransmissions for QUIC. For the throughput, goodput and (spurious) retransmissions we used pcap files converted to json. With the use of Tshark, command line version of wireshark, we can convert the pcap files captured with tcpdump to json. With QUIC being encrypted, we needed the TLS keys to be able inspect the QUIC packets. The issue we had was that some packets generated decryption errors and were not decrypted, sometimes all the packets in the file could not be decrypted. With QUIC not being finished and changes still happening, we had to use the latest build of tshark to analyse QUIC packets. The latest build can have bugs and the failed decryption of packets could be the result of one. In order to have the smallest amount of impact from decryption errors, we rerun tests if they generate a large amount of errors.

So we have identified several issues with our testing setup, parts due to the simulators and parts due to using an unstable version of Tshark. Going forward we need to take these into account, we know what the issues are and how they can affect the results. For example, we will have to analyse the results of each simulator separately, since they generate different results. However, if we notice the same behaviour happening in both simulators we can attribute that behaviour to the congestion controller and not the simulator.

5 New Reno and loss recovery

Let us now look at how congestion control and loss recovery works differently in QUIC compared to Linux TCP. We will go over the differences in design and compare the performance of the implementations. We will also uncover the reasoning behind the changes in design for QUIC. We will start first with looking at congestion control before moving on to loss recovery, where most of the changes are.

5.1 congestion control

In TCP the congestion window value is tracked in amount of packets, this can bring forth a number of issues. We will look into these possible issues and see if they're still present in the current version of TCP in Linux.

A first potential problem deals with under utilising the connection, where the amount of data that is sent is quite low compared to the available CWND. Packets with only a few bytes of data can be sent, which increases the CWND at the same rate as when larger packets are sent. With network buffers having a capacity calculated in bytes, means that it cannot handle the same amount of packets regardless of packet size. When regular buffers are filled, packets that arrive will be dropped which can be described as tail-loss. Since New Reno uses loss as a signal of congestion, the CWND will be reduced in order to drain the buffers.

Some buffers are managed differently as part of an Active Queue Management (AQM) algorithm. AQM can either drop packets that are in front of the queue when the buffer is full or when the buffer is filled over a threshold, thus allowing loss recovery to activate earlier and prevent the buffer overflowing too much. However, this cannot prevent congestion issues when a sudden burst of large packets are sent. In Linux this can still happen due to not taking in to account the amount of data that has been sent.

Another potential issue has to do mostly with how TCP acknowledges new data. TCP generates new sequence numbers based on the size of the packets. As an example let us say that the last sequence number used was 1000 and the packet size was 1000 bytes, the next sequence number would be 2000. TCP will also acknowledge data based on these sequence numbers. The issue would happen when TCP increases its CWND when new data is acknowledged, even when the entire packet is not acknowledged. Looking at the previous example, an acknowledgement could be sent containing the sequence number 2000 to acknowledge the entire packet. It is also possible to send ten acknowledgements where each time 100 bytes would be acknowledged, e.g. 1100, 1200, 1300 ... . However this is solved in Linux where it is examined if the amount of data acknowledged is equal to the size of a packet.

There is a different method that is proposed to manage the congestion window which is called Appropriate Byte Counting[26] (ABC). Rather than keeping track of the amount of packets that can be sent in the CWND, it tracks the amount of bytes. This already can help reduce the issue of under utilising the connection. Since the CWND determines how much data in bytes can be sent, sending a large amount of small packets would not increase the CWND with the same pace as when larger packets are sent. It would also prevent the second issue where splitting up acknowledgements for the same packet, increases the CWND by one packet for each ack in slow start. However in Linux this was not implemented due to ABC also imposing a growth limit per ack[26]. What can also happen, is that an acknowledgement for a large amount of data, could increase the CWND based on that amount of data all at once. This could be caused with delaying acknowledgements, where an endpoint waits for large amount of packets to arrive.

QUIC follows the recommendations described in ABC, even limiting the CWND growth for when a large amount of data is acknowledged in one ack. The CWND can double in size every RTT with a limit equal to the initial CWND of 10 packets per acknowledgement. It will add the packet size in bytes to the CWND on receiving its ack in slow start. For congestion avoidance it uses the following equation:

$$congestion\_window += \frac{max\_datagram\_size * acked\_packet.size}{congestion\_window} \tag{1}$$

QUIC decided to also recommend using a packet packet which would limit the amount of data that can be sent in a time period, since a bursts of packets can still happen when using a simple grow limit[17]. If an implementation uses a pacer, the CWND grow limit will not be needed since the pacer will prevent a large burst packets being sent. An example would be when ACK compression happens and the sender receives a number of acknowledgements within a short time window. With ABC the CWND will grow with ten packets at maximum during slow start. However, this limit is imposed per acknowledgement. This does not prevent the sender to increase its CWND by 20 if two or more acknowledgements were received within in a period of 1ms. When receiving a burst of acknowledgements due to ack compression, the CWND can still double in size and cause a large burst of packets to be sent. QUIC either requires the use of pacing which will limit the send rate when a large amount of packets can be sent or limit the CWND growth per acknowledgement similar to ABC. The other advantages of using a packet pacer, have been described in Section 3.2.6.

5.1.1 initial testing

We have performed initial testing to see how well the QUIC congestion control performs compared to TCP, where we will mainly focus on the high-level metrics: throughput, RTT and loss events. The reason to include loss events is because NewReno will see loss as congestion and will lower its CWND, this results in a lower throughput. We want to see the effects of sending data over different types of networks. Where delay, bandwidth and queue size are different.

[17] `https://github.com/quicwg/base-drafts/issues/3094`

We used three different delays: short (5ms), medium (10ms) and high (50ms). Using the three different delays, we can see how delay can affect the congestion controller. This is mainly for dealing with loss detection, where the higher the delay is the longer it takes for loss to be detected. We used the three delays combined with two different bandwidths: low (5 Mbps) and medium (10 Mbps). The difference in bandwidth is done to see how in combination with queue, it can affect the overall throughput of the connection. Will the implementations differ much from the available bandwidth, is what we will be finding out. The queue size will be configured based on the BDP. Such a short queue size is used to emulate the presence of other connections, which divides up the total queue size over all the connections. With this configuration the total amount of data that can be in-flight is two times the BDP.

We will first discuss the results of the tests from the NS-3 simulator, where first the bandwidth of 5 Mbps is used with all the different delays. After that we can compare those results to using 10 Mbps in NS-3. Next we will look at the same tests but performed in Mininet, to see if we do see similar results when comparing 5 Mbps to 10 Mbps. We have repeated each test five times to see if we got similar results, we then show the result for each implementation that has the median throughput.

We expect QUIC implementations to achieve a higher throughput, since the QUIC CWND grows at a constant rate due to using amount in bytes compared to TCP which uses amount in packets. RTT values might be lower for TCP since it runs in the kernel and whilst for QUIC data will need to passed on to user space and back. We will first look at using 5 Mbps bandwidth within the NS-3 simulator

Impl.	Throughput (Kbps)	RTT(ms)	RTTVar(ms)	#loss events
5ms, 5Mbps, Queue: 2 packets				
Aioquic	4251.95	15.61	3.5	160
Quant	4490.05	16.41	1.41	92
TCP	4200.31	14.35	6.18	92
10ms, 5Mbps, Queue: 5 packets				
Aioquic	4551.51	28.34	6.58	66
Quant	4643.69	29.49	1.65	41
TCP	4579.47	27.39	6.71	37
25ms, 5Mbps, Queue: 12 packets				
Aioquic	4587.66	63.3	10.71	71
Quant	4609.22	64.89	1.5	40
TCP	4511.08	63.79	6.09	39
Impl.	Throughput (Kbps)	RTT(ms)	RTTVar(ms)	#loss events

Table 6: Initial performance of three implementations with 5Mbps bandwidth in NS-3 simulator

One of the first things we can see in Table 6, is the increased amount of loss events for

Aioquic with all the different delays and queue sizes. So Aioquic is doing something which increases the amount of loss events, the primary cause of loss is congestion in this case. When looking at the qvis sequence diagram in Figure 27, we noticed that Aioquic would send a packet that is smaller than 1280 bytes occasionally, increasing the bytes in-flight by a small amount. Quant on the other hand only sends packets that have size of 1280 bytes. With packet queue being determined by amount of packets, this causes more loss when the queue size is small.

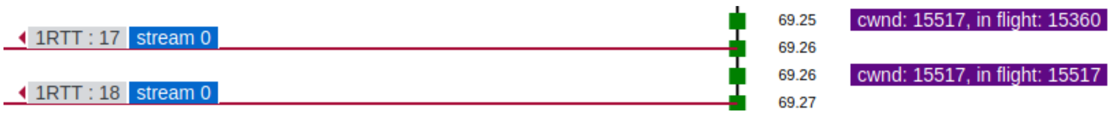


Figure 27: qvis sequence diagram showing the second packet only increasing the bytes in-flight by smaller value in Aioquic

We also see that Aioquic server sends acknowledgements to the client in a separate packet (Figure 28). The Aioquic server does this in response to a client packet containing a ping frame. So the Aioquic client also uses loss detection on its acknowledgements, though the loss events do not generate retransmissions. However, with the client not receiving any acknowledgements from the server, it sends a ping frame to trigger one. This packet with a ping frame, also contains an acknowledgement for new data. We will describe this method of loss detection in Section 5.3. With Aioquic sending another extra packet, next to the smaller data one, more loss events are generated.

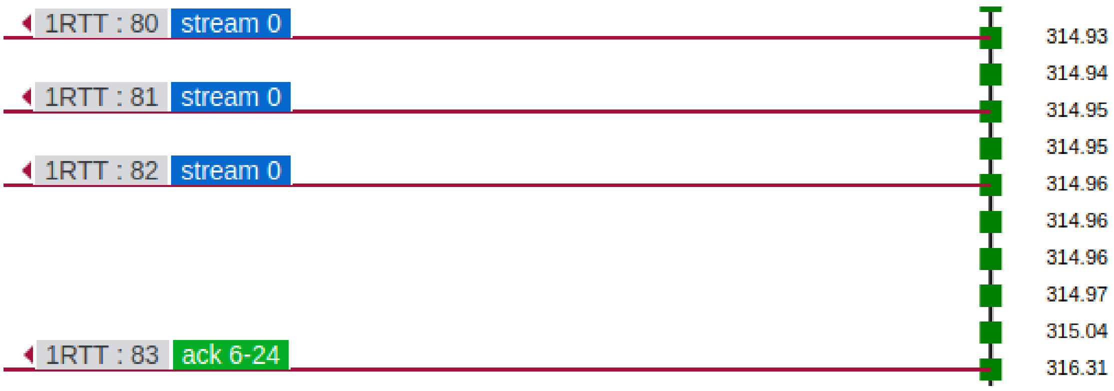


Figure 28: qvis sequence diagram showing that Aioquic server sends ACK frame in a separate packet

When looking at the throughput, we see that Quant is always the highest. Comparing Quant to TCP we notice that the CWND grows at a constant rate, shown in Figure 29. The image also shows how TCP stays at a CWND for a while and increases the CWND in steps, whilst with Quant it increases at every acknowledgement. This due to QUIC counting the CWND in bytes, whilst with TCP it needs to wait for a full CWND being acknowledged. Let us assume that TCP has CWND value of 20, TCP will only increase the CWND if at least 20 packets have been acknowledged using the CWND of 20 packets.

When comparing TCP to Aioquic, we see a similar throughput even though Aioquic experiences more loss. With Aioquic, similar to Quant, the CWND can grow at every time an acknowledgement is received, unlike TCP which waits for some time.

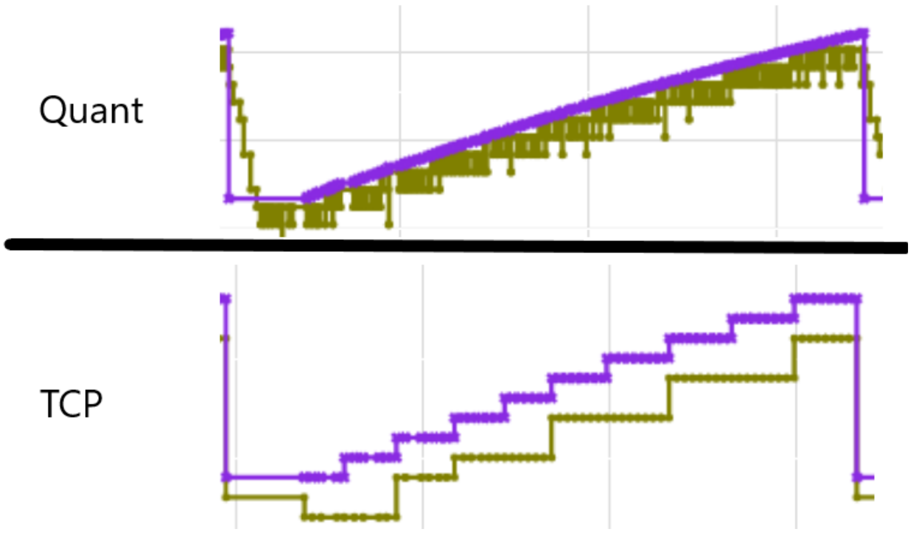


Figure 29: CWND increasing during one congestion avoidance phase in NS-3 (10ms, 5Mbps)

As we expected, we see that TCP is experiencing lower RTTs than the QUIC implementations, though the difference is not large. We do see that the RTT variance for TCP is quite large compared to Quant. When looking at how in Linux the RTT metrics are calculated, we noticed that with retransmissions the send time was not updated[18]. When the acknowledgement for the retransmission is received[19] and the latest_RTT is calculated, it uses the original send time which results in a RTT value of two times the actual RTT.

We also see that Aioquic's RTT variance increases when the delay increases, whilst Quant stays stable. One of the things we expected is that with Aioquic being written in Python, it might not perform as well as Quant which is written in C. In Table 7 we can see two RTT values for each implementation, this was for packet number 42. We used Wireshark to see what the RTT was for this packet and compared it to the qlog value and see how much extra delay was measured in the implementations. We can see that Aioquic has a much higher RTT value compared to the Wireshark output. We see this happen for multiple packets where Quant measures lower delays than Aioquic, close to the observed delay in Wireshark. Having a QUIC implementation written in Python has its limits and the RTT values less accurate than in C.

In general we see that difference between the configured RTT and the measured RTT increases, when the configured RTT increases. This due that we also increase the queue

[18] https://elixir.bootlin.com/linux/v5.4.40/source/net/ipv4/tcp_output.c#L2897
[19] https://elixir.bootlin.com/linux/v5.4.40/source/net/ipv4/tcp_input.c#L3070

Impl.	RTT(ms) Wireshark	RTT(ms) qlog
25ms, 5Mbps, Queue: 12 packets		
Aioquic	53	78
Quant	54	63

Table 7: Measured RTT for packet with number 42: Wireshark value compared to qlog value

size to have one BDP worth of queuing. With a large queue, packets will spend on average more time in the queue, which is also included in the RTT measurements.

We have seen some differences in performance when using a bandwidth of 5 Mbps, particularly with queuing where Aioquic sends more packets in a burst compared to Quant. Let us see how using a higher bandwidth will impact the performance of the congestion controllers using NS-3. The higher bandwidth of 10 Mbps will drain the queue faster, increasing the chance for Aioquic's extra packets to be queued and not get dropped. It should also reduce the queuing delay, so the measured RTTs values should be lower compared to a bandwidth of 5 Mbps.

Impl.	Throughput (Kbps)	RTT(ms)	RTTVar(ms)	#loss events
5ms, 10Mbps, Queue: 5 packets				
Aioquic	8542.83	15.01	3.37	55
Quant	8715.86	15.02	1.15	48
TCP	9016.94	13.74	3.31	52
10ms, 10Mbps, Queue: 10 packets				
Aioquic	8860.08	27.19	5.02	49
Quant	8906.9	27.02	1.39	48
TCP	9035.56	26.04	3.46	44
25ms, 10Mbps, Queue: 24 packets				
Aioquic	8870.66	60.88	8.7	64
Quant	8513.02	59.28	1.47	60
TCP	8587.65	63.69	3.15	53
Impl.	Throughput (Kbps)	RTT(ms)	RTTVar(ms)	#loss events

Table 8: Initial performance of three implementations with 10Mbps bandwidth in NS-3 simulator

Using a higher bandwidth has not changed that Aioquic has more loss events, difference is quite smaller. We do notice that the increased bandwidth has positive impact for Aioquic, where the extra packets it sends (last bit of CWND) are able to be queued. As we have seen in Section 4.4 where we compared the results of using two different bandwidths, the higher bandwidth allows packets to be removed from the queue faster.

We also see that the amount of loss events is lowest at 10ms delay and increases again at 25 ms delay. Looking at the qlogs we can see that the majority of loss happens during slow start. This is due to how fast the CWND grows in that period, it doubles in size every RTT. When it reaches a size where the queue is overflowing and packets start being dropped, there is still a delay for when the server sees that loss. We are using a First-in First-out (FIFO) queue, so packets are being dropped at the tail-end of the queue. However, there are still packets in the queue and in the other direction there are acknowledgements underway of packets that just arrived at the client. These acknowledgements and the future acknowledgements for the queued data packets, will still allow more data to be sent again and increases the CWND. The network will continue to be in this congested state, where the new data packets are being dropped until the queue has been drained enough that a gap in packets is finally detected.

We also see that at 10ms delay we have less loss than with using a 5ms delay, even though a larger queue is used. The initial CWND has a size of ten packets for QUIC as well as for TCP. At 5ms delay we only have a queue of five packets, which is not large enough to capture all ten packets. Even with using a higher bandwidth of 10 Mbps, not all ten packets can be queued and so the CWND will not grow that large since loss will be detected early.

What also happens at 25ms delay is that Aioquic now achieves the highest throughput. When looking at Figure 30 we can see how each implementations starts sending data again, after they have exited slow start and reduced their CWND. We can see that Quant sends data in larger bursts in the same time interval, which causes the buffer to overflow faster. Quant will re-enter recovery much faster than Aioquic, and so Quant will quickly have a smaller CWND than Aioquic. This allows Aioquic, which has a larger CWND, to send more data and thus generate a higher throughput.

Lastly we suddenly see that TCP experiences a high average RTT than the QUIC implementations. Looking at Quant, when it enters the first recovery period it sends out retransmissions in large bursts. After that it continues to send packet in bursts. The problem with sending packets in bursts that queue fills up very quickly, since it removes packets at a steady rate which depends on the configured bandwidth. Thus Quant will lower its CWND earlier than TCP, due to the bursts of packets causing loss faster. However, TCP also has an higher average RTT compared to Aioquic.

Comparing the average RTT in Aioquic compared to TCP at 25ms delay, the difference there is that the CWND grows continuously in Aioquic. With Aioquic also fully utilising the CWND, it will start to send more packets due to the leftover bytes in the CWND. Thus the queue fills up faster with Aioquic, causing loss earlier than in TCP. With TCP spending more time with a high CWND, it causes that more packets will experiences longer queues and thus longer queuing delay. With the QUIC implementations spending less time at a high CWND, it sends less packets that will experience the high queuing delay.

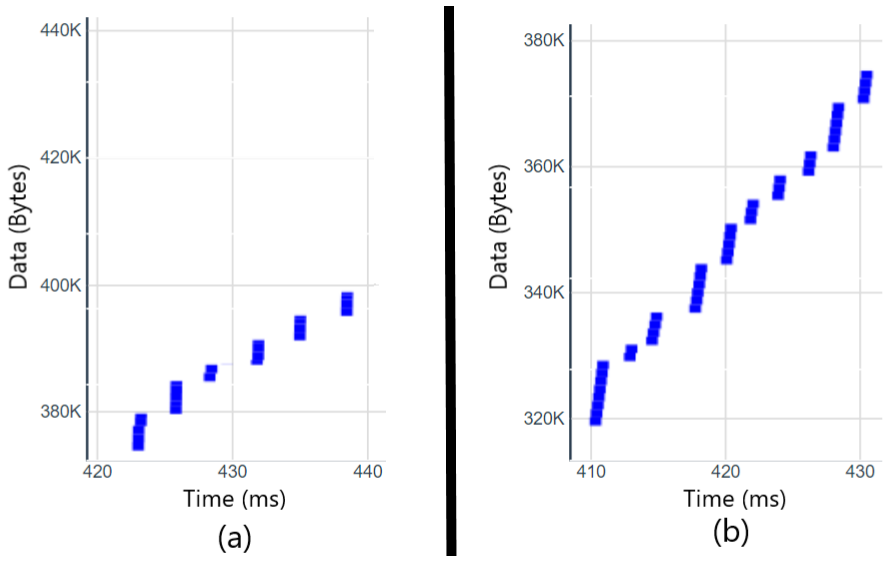


Figure 30: Quic implementations sending data (blue) after exiting slow start, Quant (b) sends larger burst of packets than aioquic (a)

Let us now look at the test results of the Mininet simulator and see if we notice the same patterns happening that we seen with NS-3. We will first start with using a bandwidth of 5 Mbps.

Looking at the results in Table 9, we can see a similar pattern happening with Aioquic experiencing the most amount loss in the first two scenarios. This is again due to the extra packet that Aioquic sends in order to fully use the available CWND.

At 5ms delay Quant has more loss than TCP due to the delayed acks, where an ack arrives for every two packets. This means that two packets will be acked at the same time, which causes Quant to send two packet out at the same moment during congestion avoidance. The queue can handle it until the CWND has grown enough to send a third packet. With TCP we see that most of the time only one packet is acknowledged at a time, causing TCP to only send one packet back during congestion avoidance. If the CWND grows by one packet, TCP would then send two packets wich the queue can handle.

TCP has more loss events at 10ms due to acks arriving every two packets or more received, causing large burst to be sent, up to four packets at once. This is similar to why Quant experience more loss earlier.

We have seen the results for a bandwidth of 5 Mbps, let us now look at the results when using a bandwidth of 10 Mbps. Like with NS-3, we should see a reduction in loss events due to the queue being drained faster. With less loss happening, we should see an improvement in throughput for all implementations.

Impl.	Throughput (Kbps)	RTT(ms)	RTTVar(ms)	#loss events
5ms, 5Mbps, Queue: 2 packets				
Aioquic	1929.7	12.16	0.82	1899
Quant	1963.87	12.06	0.21	857
TCP	2039.07	10.12	4.72	129
10ms, 5Mbps, Queue: 5 packets				
Aioquic	1937.84	21.95	1.32	566
Quant	2149.13	22.29	0.66	201
TCP	2271.22	21.78	21.18	208
25ms, 5Mbps, Queue: 12 packets				
Aioquic	1988.42	52.26	1.62	108
Quant	1987.45	53.49	1.13	115
TCP	2402.91	50.73	12.12	52
Impl.	Throughput (Kbps)	RTT(ms)	RTTVar(ms)	#loss events

Table 9: Initial performance of three implementations with 5Mbps bandwidth in Mininet simulator

Table 10 shows us the results of using a bandwidth of 10 Mbps in Mininet, showing a reduction in loss events when comparing to the 5 Mbps results. What we see happening is that TCP still achieves the highest throughput. With the congestion window growing in steps, this allows TCP to send more data before loss is detected. The result of this is that TCP undergoes less recovery periods and has to perform less retransmissions.

We do see for TCP quite a large reduction in throughput when the delay is increased to 25ms from 10ms. Even though TCP has less recovery periods, they now take more time due to the longer delay. Rather than waiting for 20ms for a retransmission to be received and acknowledged, it will now take at least 50ms. With having a larger queue, the retransmission will also experience longer queue delay than using the smaller queue at 10ms delay.

We can see that loss detection is a very important part of the NewReno congestion controller, where we see the throughput being affected by how much loss is detected and how many times the algorithm goes into recovery. Let us focus now on the loss detection methods used in TCP and QUIC, and see how they perform.

Impl.	Throughput (Kbps)	RTT(ms)	RTTVar(ms)	#loss events
5ms, 10Mbps, Queue: 5 packets				
Aioquic	3640.17	11.78	1.17	561
Quant	4090.16	11.88	0.58	270
TCP	4526.47	10.92	10.61	223
10ms, 10Mbps, Queue: 10 packets				
Aioquic	4024.19	21.86	1.2	141
Quant	4013.16	22.31	0.85	145
TCP	4933.38	20.45	7.03	55
25ms, 10Mbps, Queue: 24 packets				
Aioquic	3835.43	52.03	1.4	52
Quant	3964.52	52.44	0.85	76
TCP	4481.28	50.6	6.69	39
Impl.	Throughput (Kbps)	RTT(ms)	RTTVar(ms)	#loss events

Table 10: Initial performance of three implementations with 10Mbps bandwidth in Mininet simulator

5.2 Loss recovery

Loss recovery for QUIC has more differences compared to Linux TCP, with the main differences being the methods of loss detection. We will first focus on using timestamps and RTT measurements to detect loss.

5.2.1 Time threshold calculation

When looking at the equation used by TCP in Linux and QUIC for the time threshold, there are differences. In Linux, TCP uses the RACK loss detection method, described in Section 3.2.5. When looking at the code, we do not see any clear differences to the RACK draft. The important part of RACK, is using an adaptive threshold, which increases if spurious retransmissions are detected.

QUIC uses a different equation to calculate the time threshold. It was first known as Early Retransmit and the equation has changed over time. Early Retransmit[27] was introduced when loss would occur in situation where the CWND is very small due to a low BDP network or because TCP is being application limited. The basic duplicate ack method requires at least three duplicate acks to be received before a packet can be declared lost. In the situations described earlier where CWND is small, this makes it difficult to even send three duplicate acks. An example would be when three packets are sent and the first is lost, this would only generate two duplicate acks. What would follow is then a retransmission timeout which could take at least one second to trigger, it would also reset the CWND back to its initial value.

Early retransmit allows the duplicate packet threshold to be reduced when there are a maximum of four packets in-flight and the sender cannot send any data. The new threshold value would be one less than the amount of packets in-flight: e.g. if there are three packets in flight, the duplicate threshold would be two. However, the threshold of three packets was chosen to prevent spurious retransmissions due to reordering. By lowering the threshold, early retransmit would send more spurious retransmissions. Researchers saw that in Linux a timer was used, which delays the marking of lost packets during early retransmit. This delay would allow acknowledgements to arrive for the other packets, thus preventing spurious retransmissions from happening. This delay method was then introduced into QUIC as well[20].

The early QUIC drafts allowed a time-based method, to either replace the packet threshold method or to work along with it. With early retransmit using a similar time threshold compared to the time-based method, it was decided to remove the early retransmit to make loss detection less complicated. This change causes QUIC to have both a packet number and time threshold based loss detection methods. The equation for the time threshold has changed over time, since draft 17 it is this:

$$time\_threshold = max(kTimeThreshold * max(smoothed\_rtt, latest\_rtt), kGranularity) \tag{2}$$

This is quite different from TCP where we do not see a dynamic reordering window. In this equation *kTimeThreshold* has a value of 9/8 and *kGranularity* has a value of 1ms, which is used to prevent declaring packets lost too early in low delay environments. QUIC changed the threshold from $\frac{1}{4}$th extra to $\frac{1}{8}$th extra, where the $\frac{1}{4}$ threshold was used in Linux for the Early Retransmit timer[21]. The time difference in actual arrival time and the expected time of arrival compared to the minimum RTT, would mostly be $\frac{1}{8}$th of the RTT or lower (Figure 31). So if packets are sent over a different path, the difference in latency is quite low. So lowering the threshold would still prevent detecting spurious loss for most reordering events. Using the $\frac{1}{8}$ threshold still caused less spurious retransmissions than the duplicate ack threshold loss detection[22]. With large BDP networks and a lot of packet being in-flight, three packets with a higher packet number can be received quite fast which can quickly cause three duplicated acks.

Compared to Linux TCP which only uses the latest_RTT, QUIC will either use the smoothed_RTT or the latest_RTT, depending on which value is the largest[24]. First there is the situation where the latest_RTT is the lowest of the two parameters. This could happen when there is reordering. The packet which was sent last could be going

[20] `https://tools.ietf.org/html/draft-ietf-quic-recovery-08#page-7`
[21] `https://elixir.bootlin.com/linux/v4.7.10/source/net/ipv4/tcp_input.c#L2005`
[22] `https://mailarchive.ietf.org/arch/msg/quic/n4i5xMmIxmBvnakWIqmBQHFY5ko/`

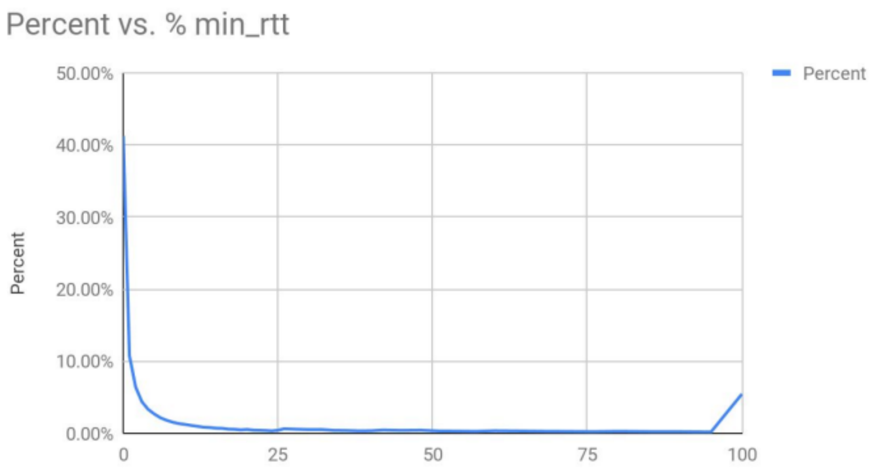


Figure 31: Arrival time difference as fraction of min_rtt SOURCE: `https://datatracker.ietf.org/meeting/102/materials/slides-102-maprg-udp-packet-reordering-ian-swett-00`

over another path which has less queuing. Using the latest_RTT here could cause spurious loss detection for the packets that were sent earlier, which are experiencing a longer delay due to more queuing. Secondly, there is the possibility of the latest_RTT increasing due to increased queuing. The smoothed_RTT does not immediately catch up when there is a sudden burst of packets sent, that increases the queuing drastically. Using the smoothed_RTT could detect loss spuriously for the packets that experience this increased queuing delay.

5.2.2 All ack-based loss detection methods

When looking in Linux to see which loss detection methods were used, we noticed mentions of RACK and TCP scoreboard. Looking at TCP scoreboard[23], it is based on the SACK loss detection method which we described in section 3.2.2. When duplicate acks are being received with selective ack ranges, the TCP update scoreboard function will be called. It will check how many packets with a higher sequence number, than the potentially loss packet, have been acknowledged through SACKs. If that amount is higher than the reordering threshold, the packet will be marked as lost[24].

The reordering threshold TCP uses is an adaptive threshold, that is increased when reordering is detected. TCP uses SACKs again to detect reordering, it will check if a packet with a higher sequence number is sacked before a packet with a lower sequence number, that is not marked as lost and that is not a retransmission. The threshold can also be decreased when a timeout event happens and data has been sacked. If the amount

[23] `https://elixir.bootlin.com/linux/v5.4.40/source/net/ipv4/tcp_input.c#L2251`
[24] `https://elixir.bootlin.com/linux/v5.4.40/source/net/ipv4/tcp_input.c#L2251`

of packets sacked is lower than the reordering threshold, the reordering threshold will be reduced to its default of three packets [25].

We noticed that only one of the two methods is active. Default it is RACK, whilst the TCP scoreboard has been disabled. The implementation of RACK uses the methods described earlier in Section 3.2.5.

QUIC on the other hand uses two ack-based methods at the same time. The first method used is a time threshold method, which is described in more detail further in Section 5.2.1. The second one based on the well-known packet threshold based on the duplicate ack method. The QUIC method allows loss to be detected where a high amount of packet loss is present. With TCP there still needs to be at least three packets received with a higher sequence number, compared to QUIC where one packet is sufficient. Figure 32 shows an example when packets 1 to 3 are lost but 4 arrives. When the sender sees the ack for PN 4, it will mark PN 1 lost since its number is three lower than 4. The other packets are not detected as lost since they do not go over the threshold. The scoreboard method in TCP would not mark packets as lost yet, due to only one packet with a higher packet number being acknowledged. This method is similar to the method used in Forward Acknowledgement[28], where the highest sequence number is used of the packets that were acknowledged by SACK.

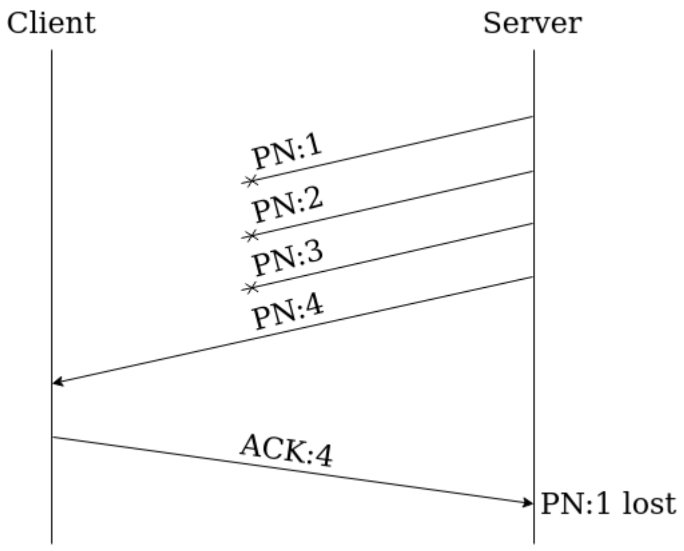


Figure 32: Packet threshold loss detection method in QUIC

QUIC uses a fixed reordering threshold of three packets rather than an adaptive threshold. As to why the recovery draft does not require adaptive thresholds for the packet and time threshold methods, it is because there is still uncertainty towards which adaptive threshold method works best[26]. The recovery draft does encourage developers to exper-

[25] `https://elixir.bootlin.com/linux/v5.4.40/source/net/ipv4/tcp_ipv4.c#L2673`
[26] `https://mailarchive.ietf.org/arch/msg/quic/pTx10oO17chq39odBm84XKH5_xY/`

iment. Through experimentation and analysing results can a good adaptive threshold method be found, which then can be included into the recovery draft.

This is also an interesting choice to both use a time threshold and packet threshold loss detection method, compared to TCP which only uses the RACK method. When looking back at when the choice was made to combine the two methods, Linux was doing the same by using the scoreboard method and RACK. Another reason for QUIC to use both methods, was that low latency networks were difficult for time based detection, because of timers values not being accurate enough[27].

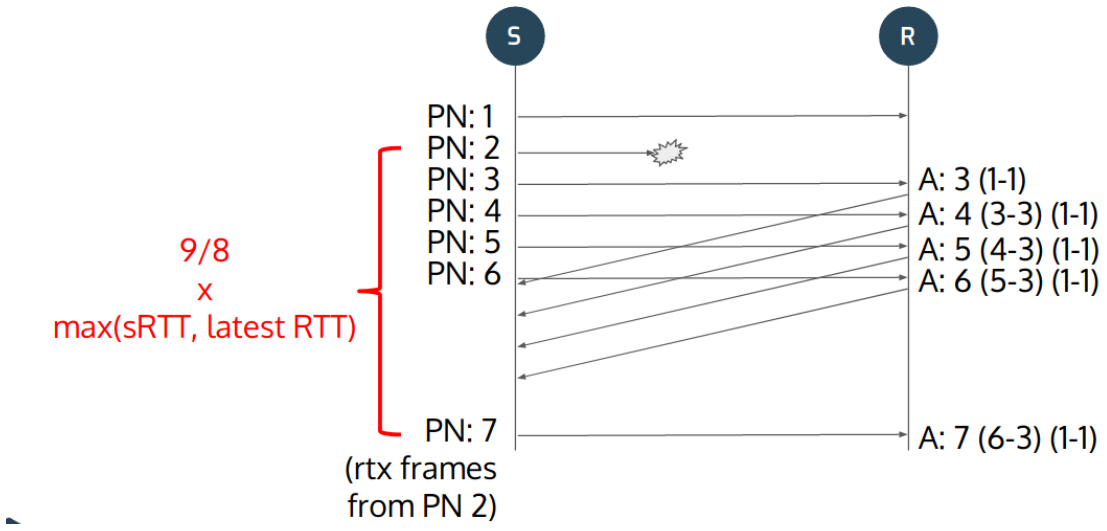


Figure 33: Time threshold loss detection method in QUIC source: `https://www.ietf.org/proceedings/103/slides/slides-103-tcpm-sessb-quic-loss-detection-congestion-control-01`

However since then TCP has changed this to only use RACK method as a default, this was changed in kernel version 4.18[28]. This was done after A/B experiments were performed by Google on their servers, where they compared RACK/TLP vs TCP scoreboard (RFC 6675) and RACK. Looking at the results the RACK/TLP method reduced the recovery latency by 10% compared to RFC 6675[29]. This means that RACK/TLP allowed TCP to detect loss faster and retransmit all the lost data faster, by exiting recovery earlier TCP could send new data earlier compared to RFC 6675.

We have asked on the mailing list the question why QUIC still uses both methods for loss detection. The answer I got has to do with the packet numbers[30]. As mentioned earlier,

[27] `https://github.com/quicwg/base-drafts/issues/1212#issuecomment-373977171`

[28] `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b38a51fec1c1f693f03b1aa19d0622123634d4b7`

[29] `https://patchwork.ozlabs.org/project/netdev/cover/20180516234017.172775-1-ycheng@google.com/`

[30] `https://mailarchive.ietf.org/arch/msg/quic/pTx10o017chq39odBm84XKH5_xY/`

QUIC packet numbers are only used to indicate the order of packets. Whilst the offset in data, to make sure the data is ordered correctly, is present in the payload. When QUIC sends retransmissions, it does so by sending a new packet with a new packet number and placing the lost payload in this new packet. This allows QUIC to use this loss detection method at all times, whilst with TCP it cannot be used during recovery since packet numbers are being reused. However, apart from that there is not much difference for when congestion controllers are not in recovery.

QUIC still uses both the packet and time threshold methods at the same time where if one method detects packet loss, QUIC will go into recovery and retransmit data. In a data center context, having the packet threshold method would be helpful since it is hard to get accurate enough timers. When transmitting over the internet where the delays are longer, would it still be necessary to use both methods. What happens when packets are being reordered? Does using both methods increase the amount of spurious retransmissions and if so, which method would cause the most?

Let us revisit the earlier tests and look at two things: which loss detection method detects the most loss and how will packet reordering affect the loss detection. For reordering we introduced this behaviour using TC Netem, where every 500th packet will not experience the configured delay and skips ahead in the queue. The reordered packet will arrive $RTT_{min}/2$ sooner, which as we have seen earlier is quite large for reordering. So we choose a high value of 500 prevent a large amount of moderate reordering happening, but still high enough to see the effect it can have on throughput and loss.

With the time threshold method using the largest value between latest_RTT and smoothed_RTT in QUIC and the reordered packet experiencing a lower RTT, we do not expect the time threshold being affected by reordering all that much. When a packet is reordered, the smoothed_RTT will be used to calculate the time threshold. The smoothed_RTT will not reduce as much and using the extra $\frac{1}{8}$th threshold should give enough time for the acknowledgements to arrive for the packets that were sent earlier than the reordered packet.

It is the packet threshold that would be most susceptible to detecting spurious loss when a reordering event happens. Due to the reordered packet skipping ahead of the queue, this causes a packet with a higher packet number to be received by the client. In a network where a lot of packets can be in-flight due to a large amount of queuing, the reordered packet can skip ahead of many packets, which can be more than the reordering threshold used.

We will go over the results in the similar order as in Section 5.1.1 where we will first look at the results in NS-3. Starting with a using a bandwidth of 5 Mbps and the three different delays: 5ms, 10ms and 25ms. We will focus on the the amount of loss events recorded in the qlog and which loss detection method was used for each loss event.

Impl.	pkt_tresh	pkt_tresh (reorder)	time_tresh	time_tresh (reorder)
5ms, 5Mbps, Queue: 2 packets				
Aioquic	16	14	144	142
Quant	0	0	92	93
10ms, 5Mbps, Queue: 5 packets				
Aioquic	11	19	55	52
Quant	1	14	38	39
25ms, 5Mbps, Queue: 12 packets				
Aioquic	62	128	5	3
Quant	38	99	3	2
Impl.	pkt_tresh	pkt_tresh (reorder)	time_tresh	time_tresh (reorder)

Table 11: Comparison of ack-based loss detection methods with and without reordering in NS-3 simulator with 5Mbps

Looking at the loss detection events from the first series of tests (Table 11), we can initially see that the time threshold method is the first one to detect loss in the first two scenarios. We can see in the qlogs that when acknowledgement is received which shows a gap in packets received by the client, the gap is most of the time not larger than the packet threshold. This allows the time threshold method to detect loss first.

However we do see that at 25ms delay and a queue size of twelve packets, that most loss is detected with the packet threshold. At this point the long delay increases the time threshold and now there is enough time for more packets to be received so the highest packet number goes over the reordering threshold.

We can also see that at 25ms both implementations experience a slight increase in amount of packets loss. The extra queue size that is available allows for more packets to be received if they are sent in a burst, for example the initial windows of 10 packets. In slow start the CWND can grow very fast the longer NewReno stays in slow start, this increase in data sent can quickly fill up the queue. With the queue being full, all the other data that is sent will be lost. With a larger CWND, the more data is send and will be lost.

When we introduce reordering we only start to see the effect of it starting at 10ms delay, where we see a slight increase in packet threshold events. However at 25ms delay, the amount of packet threshold events more than doubles. In Figure 34 we can see an acknowledgement arrive at the server, just after a reordering event. The gap between the highest acked packet number and the second highest is quite large, larger than the threshold of three packets. As we have described earlier, the reordered packet will skip the queue. With the queue also being larger now, more packets will be skipped, for example it is possible to skip ahead of eleven packets.

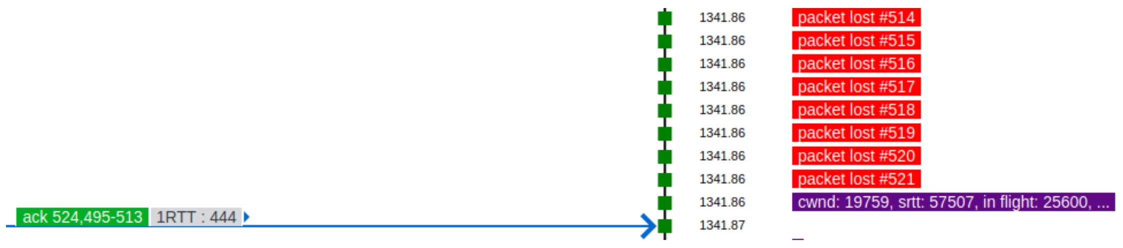


Figure 34: Quant server receives ack for reordered packet, declares eight packets lost (25ms, 5Mbps)

So we can see how reordering already affects the packet threshold loss detection, let us also look at the results of using a bandwidth of 10 Mbps with NS-3. The higher bandwidth allows the queue to be drained faster, which might have an impact of the packet threshold loss detection. Since now more packets can be received in a time interval, which might show a three packet number gap before the time threshold would detect loss. However, with reordering it should not have an impact, due to the problem being the order in which packets are received and not how far apart in time they were received.

Impl.	pkt_tresh	pkt_tresh (reorder)	time_tresh	time_tresh (reorder)
5ms, 10Mbps, Queue: 5 packets				
Aioquic	16	25	34	42
Quant	5	8	41	41
10ms, 10Mbps, Queue: 10 packets				
Aioquic	38	49	12	15
Quant	25	63	20	25
25ms, 10Mbps, Queue: 24 packets				
Aioquic	63	178	1	3
Quant	51	170	7	5
Impl.	pkt_tresh	pkt_tresh (reorder)	time_tresh	time_tresh (reorder)

Table 12: Comparison of ack-based loss detection methods with and without reordering in NS-3 simulator with 10Mbps

As we suspected we can see in Table 12 that the packet threshold detects more loss at 10ms delay now, compared to using a bandwidth of 5 Mbps. With the queues being drained faster, a gap of three packets or more is discovered earlier than the time threshold timer. We have seen this before when analysing the effect a higher bandwidth has in Section 4.4. We can also see that the impact of reordering increases when the queue size is increased, due to the reordered skipping more packets which are in the queue.

When looking the test results with 10ms delay, Quant shows a larger increase in packet threshold loss events when compared to Aioquic. With Quant having more accurate RTT timings, it can detect loss earlier with the time threshold method. An observation we have made in Section 5.1.1.

We will now look at the results of using a bandwidth of 10 Mbps with the Mininet simulator. The results of 5 Mbps bandwidth, showed not much difference when introducing reordering. Due to the queuing in Mininet, less packets will be in-flight which does not trigger the packet threshold loss detection when a packet is reordered. So we will skip those results and look at the 10 Mbps bandwidth results and see if we observe an increase in packet threshold loss detection with reordering.

Impl.	pkt_tresh	pkt_tresh (reorder)	time_tresh	time_tresh (reorder)
5ms, 10Mbps, Queue: 5 packets				
Aioquic	364	357	198	220
Quant	10	13	265	271
10ms, 10Mbps, Queue: 10 packets				
Aioquic	96	101	52	56
Quant	1	22	145	132
25ms, 10Mbps, Queue: 24 packets				
Aioquic	41	98	12	16
Quant	0	71	76	64
Impl.	pkt_tresh	pkt_tresh (reorder)	time_tresh	time_tresh (reorder)

Table 13: Comparison of ack-based loss detection methods with and without reordering in Mininet simulator with 10Mbps

When we look at the results (Table 13) without reordering we do notice that Aioquic has a lot more packet threshold loss events compared to Quant. Event at 25ms delay, Quant has detected loss purely with the time threshold method. This is the result of having more accurate RTT measurements, as described earlier, as well as how Mininet actually has less queuing space (Section 5.1.1).

We see however, that Quant experiences more loss at 25ms delay, we also noticed that Quant achieves a higher CWND during slow start. With Aioquic loss is detected earlier due to the extra packet being send which contains the leftover of the CWND. With the CWND being smaller, the amount of packets lost is less compared to Quant.

We do see, similarly to NS-3 tests, that reordering has a large impact on the packet threshold method. Certainly when the queue and delay increases, we see large increase with the packet threshold loss events.

Like we suspected, the packet threshold method is susceptible to packet reordering where a packet skips the queue and arrives earlier than expected. So we decided to repeat the test with the configuration of: 25ms delay, 10 Mbps bandwidth and the queue size of 24 packets. We chose this specific configuration because we noticed the largest amount of spurious retransmissions for both implementations and simulators. However, we did configure both implementations to first use the default packet threshold of three packets, and then also an increased threshold of six and nine packets.

We decided to work with small increases in the threshold to see if it already shows a signification amount of improvement. We might expect an increase in time threshold loss events, due to the higher packet threshold. Where the higher packet threshold takes more time since it needs packets with an even higher number. Similarly to Linux, where the reordering window threshold is slightly increased, we want to see if it is worth it working with slightly higher thresholds. If the amount of loss events detected with the packet threshold is still higher when using a higher threshold, perhaps it is better to just work with the time threshold.

Impl.	pkt_tresh = 3	pkt_tresh = 6	pkt_tresh = 9
25ms, 10Mbps, Queue: 24 packets			
NS-3			
Aioquic	178	157	133
Quant	170	143	116
Mininet			
Aioquic	98	58	41
Quant	71	62	35
Impl.	pkt_tresh = 3	pkt_tresh = 6	pkt_tresh = 9

Table 14: Comparison of packet threshold loss detection with different thresholds

In Table 14 we can see the results from both simulators where we have increased the packet threshold value. When comparing the results, we do see a decrease happening in the amount of loss events detected by the packet number threshold. However, when inspecting the qlogs we still noticed that loss was being detected spuriously. Where for example, more than nine packet numbers were skipped by the reordered packet. This still resulted in a CWND reduction and a lowering of the throughput. When inspected the time threshold loss events, we did see a slight increase as expected.

Using an increased packet threshold did show a reduction in spurious loss events, but reordering would still cause packets to be detected as lost by the packet number threshold method. Perhaps using only the time threshold, like with Linux, might prove to be a better alternative. So we repeated the test with the same configuration with only using the time threshold method: 25ms delay, 10 Mbps bandwidth and queue of 24 packets.

25ms, 10Mbps, Queue: 24 packets			
Impl.	total with both methods	time_thresh = 1/8	time_thresh = 1/4
NS-3			
Aioquic	181	99	94
Quant	175	72	74
Mininet			
Aioquic	114	49	49
Quant	135	73	71
Impl.	with both methods	time_thresh = 1/8	time_thresh = 1/4

Table 15: Overview of retransmissions with only time threshold enabled compared to total loss events when using default setting

Looking at the results in Table 15, we see clearly a large reduction in loss events. However, the amount of loss events generated when only using the time threshold method, is still higher compared to having no reordering. We suspected that the time threshold method would some how detect some reordering as loss. So we repeated the test whilst using an extra time threshold of $\frac{1}{4}$th, compared to the original of $\frac{1}{8}$th.

However, we do not see a noticeable decrease in the amount of loss events. When inspecting the qlog we did not see any loss being detected when a reordering event took place. We then looked at slow start, where we would see the most loss happening due to how fast the CWND grows. We compared the results of only using time threshold loss detection with using both methods without reordering. What we saw happening is that the congestion window would grow larger when only using time threshold loss detection, we also saw that more packets were being sent out. We also noticed that the packet threshold loss detection would detect all the loss during slow start when using both methods.

So what is happening is that with only using time threshold, loss is being detected later than using packet number loss detection. We can see that when using both loss detection methods, the packet threshold method is always first. For the NewReno congestion controller, loss is the signal for congestion, only then is the CWND reduced. With loss being detected later than normal, the CWND stays high where it continues to send too much data which overflows the queue. The queue however, is still being drained at a steady rate, which allows some packets to be queued. The acknowledgements for these packets allow the more data to be sent and grow the CWND even larger.

So in general when delaying the loss detection, it keeps the network in a congested state for a longer period of time. In that state the majority of packets send by the server, will be lost. This will have an impact of performance since more packets will need to be retransmitted.

5.2.3 exit recovery state

Using the loss detection methods, both protocols can step into the recovery phase if packets are marked as lost. Both QUIC and TCP enter recovery at the same time, however the exit criteria are different. TCP exists recovery when all lost packets were acknowledged that were sent before going into recovery [31]. This was described in Section 3.2.1, which is what the NewReno algorithm does.

QUIC will exit as soon as one packet is acknowledged that was sent after going into recovery. Since QUIC handles retransmissions differently due to using new packet numbers, this means that at least one retransmission was received. This means that QUIC can enter recovery again if some later retransmission was not received. It was decided to use this method to allow QUIC to react more rapidly when large amount of congestion is happening[32]. This can greatly reduce the amount of retransmissions when the CWND is very large. When after the first reduction, the CWND is still too large and causes another large amount of retransmissions. So with QUIC, the new CWND will be used for at least 1RTT, which is the time it takes for the first retransmission to be acknowledged. A question, is using the 1RTT threshold long enough to prevent QUIC going into recovery again when it is not needed. If a short burst of traffic is sent around the same time the first retransmission is received, it can cause other retransmissions to be lost. But the new CWND size could be small enough to allow the queue drain, though QUIC will reduce its CWND again due to a retransmission being lost.

When revisiting the results of the 25ms delay, 10 Mbps bandwidth and queue size of 24 packets, we see that TCP only has one long recovery period. Figure 35 shows that TCP only has one CWND reduction (purple line) when performing slow start during the first second of the connection. Looking at Quant for example (Figure 36), we can see that the CWND decreases two times in a short interval. The same behaviour was also observed in the Mininet simulator. When the second recovery period starts and the CWND is reduced again, the size is $\frac{1}{4}$th of CWND at the end of slow start. This reduces the throughput even more compared to TCP, which for short flows is not fair.

With TCP delaying another CWND reduction, it could be that when QUIC is competing with TCP, QUIC might reduce its CWND whilst TCP is still in recovery. When TCP exits recovery it will have a much higher CWND than QUIC and will be able to send more data than QUIC. TCP could continue whilst using more resources than QUIC for the entire connection. We have not been able to test the impact on fairness, since we did not have enough time to set up a fairness test between Linux and the QUIC implementations.

[31] `https://elixir.bootlin.com/linux/v5.4.40/source/net/ipv4/tcp_input.c#L2820`
[32] `https://mailarchive.ietf.org/arch/msg/quic/YO8k1KK7ywCAaNymbwJmxEmL8Cg/`

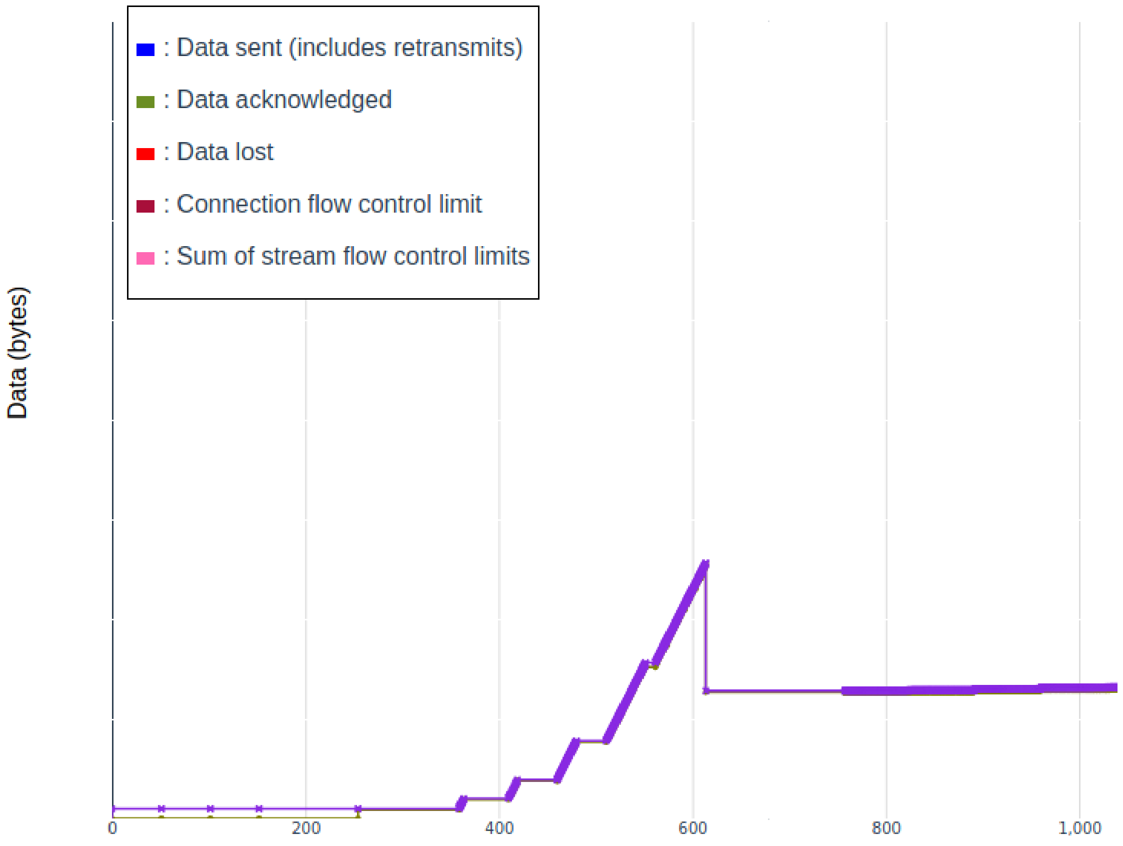


Figure 35: TCP's long and single recovery period in interval of 1 second in NS-3, CWND as purple line

5.3 Probe timer and Retransmission timer

A difficult situation could happen when suddenly no acknowledgements are being received. In this scenario it is import to still be able to detect loss. However, we see that TCP and QUIC focus on only using the ack-based methods for loss detection. In a situation where no acknowledgements are being received, both TCP and QUIC use a similar method to trigger acknowledgements.

Looking at Linux there are two different timers used for when ack-based methods should fail, mostly to detect tail-loss. First there is the classic Retransmission Timeout (RTO), which will be triggered if no acknowledgement is received for some time. TCP will treat this as severe congestion and will reduce the CWND back to the initial value, it also refreshes the SStresh parameter. After completing the recovery phase, TCP will continue on with slow start again until the CWND reaches SSthresh or packet loss is detected.

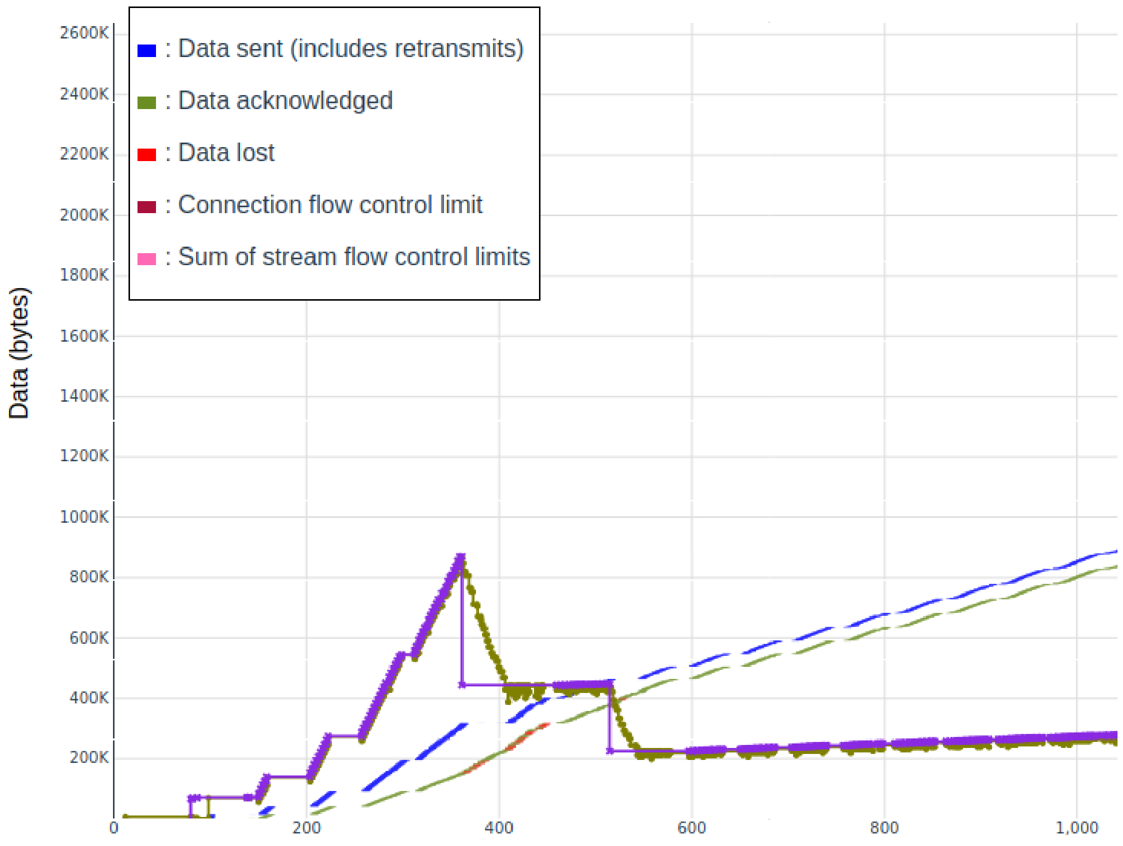


Figure 36: Quant's two shorter recovery periods in interval of 1 second in NS-3, CWND as purple line

Secondly there is the Tail-Loss Probe (TLP) which was introduced to handle tail-loss faster, since the RTO is on a long timer. We have discussed this algorithm in greater detail in Section 3.2.5. Linux uses two different equations depending on the state of the connection[33], with the first one being the following:

$$timeout = 2 * smoothed\_rtt + RTO\_MIN \tag{3}$$

Equation (3) is used when there is only 1 packet in flight, RTO_MIN has a value of 200ms. As for when there are more packets in flight, the following equation is used:

$$timeout = 2 * smoothed\_rtt + TIMEOUT\_MIN \tag{4}$$

For Equation (4) $TIMEOUT\_MIN$ has a value of 2ms. When there is no RTT measurements yet, Linux will use a initial timer value of one second.

[33] `https://elixir.bootlin.com/linux/v5.4.51/source/net/ipv4/tcp_output.c#L2491`

When looking at QUIC, there is only the Probe Timeout (PTO) in the specification. The retransmission timeout was merged with TLP and became the PTO. Let us start first when QUIC used both the TLP and RTO and see why it changed. The first loss detection timer that would fire in QUIC would be the TLP, this is the same in Linux. However, QUIC allows for two probe timers to be fired before and RTO is scheduled.

After the second probe timer was expired and a new probe packet was sent, the RTO would be scheduled. If the RTO timer would fire, QUIC would handle it differently compared to Linux. Rather than immediately marking all packets lost and going into recovery, QUIC would repeat TLP behaviour. It will send data to the other endpoint in order to trigger acknowledgements. When an RTO has fired and an acknowledgement is received where there are unacknowledged packets, QUIC will react in a similar manner to TCP when an RTO has fired. It will reduced its CWND to the initial value and will perform slow start again.

Compared to TCP, QUIC will first wait for an acknowledgement before taking severe action by reducing its CWND to the initial value. This method is used to prevent spurious detection of loss and going into recovery unnecessary, which was described in the Slack channel of QUIC. This can happen when the RTT increases by a large amount, due to burst traffic appearing. It could also happen when data packets are arriving but the acknowledgement packets are being dropped, due to congestion in that direction. TCP has FRTO to help detect spurious loss detection and allow the CWND be reverted, but it will still have had an impact on transmission rate. FRTO in TCP will check the second acknowledgement that is received after the RTO was fired. If all the data that was marked lost due to the RTO, is acknowledged, then the RTO was spurious and TCP will continue in congestion avoidance phase.

Since the TLP and RTO in QUIC behaved very similarly, it was decided to unify the TLP and RTO into PTO. Using the two different timers made timer based loss detection more complicated. It is still possible to re-enter slow start if the network is experiencing a lot of packet loss, which we will describe later in this section. QUIC also uses a different equation for the PTO, which is the following:

$$timeout = smoothed\_rtt + max(4 * rttvar, kGranularity) + max\_ack\_delay \tag{5}$$

First QUIC includes the max_ack_delay because the RTT calculations remove this delay to get the actual network delay. By not including the ack delay the probe timer could be fired spuriously. The rest of the equation is based on RFC 6298[29].

Looking at initial tests, we could also see the average value for the probe and retransmission timers. However, results of these tests did not show many timers being fired. Towards the end of the connection, the CWND was not too large to cause loss at the end. At that point acknowledgements were still arriving and so no timer would fire. So we decided to primarily focus on the ack-based loss detection methods.

5.3.1 Persistent congestion

With the change of only using PTO as a timer, a new logic had to be implemented when dealing with a large amount congestion (similar to TCP RTO). QUIC calls this scenario persistent congestion and by declaring it the CWND will be reduced to its initial value of ten packets. When TLP and RTO were unified, the logic for declaring persistent congestion had to move to the PTO. With QUIC allowing two TLPs to be fired before an RTO, it was first decided to declare persistent congestion if three consecutive PTOs were fired. However, the issue with PTO is that it is rescheduled if new data is being sent. This can happen when QUIC is application limited, so new data could arrive in the QUIC stack from the application before the PTO would normally fire. QUIC would then send the new data in packet, for which the PTO timer is calculated and will overwrite the current timer.

The current logic switches to a time based threshold, where the difference in send time between the first and last lost packets is used. In Figure 37 we can see an example, where packet 12 is lost and no other packets are sent. The first probe (PTO1) will be sent some time later and is lost as well, the second probe is sent and also lost. Finally the third probe is received and the client responds with an ACK for packet 15. Due to the ack-based loss detection methods, packets 12-14 can be detected as lost. Persistent congestion is declared when the difference in send time for packet 12 and packet 14 is larger than three times the probe timer.

Starting in July 2020 there have been quite a few issues raised with declaring persistent congestion. The current logic can declare persistent congestion too early, an example would be in the beginning of the connection [34]. With QUIC using NewReno, experiencing loss early on in the connection is devastating. Since QUIC will exit slow-start with a small CWND and continue with congestion avoidance which grows the CWND slowly. Spuriously declaring persistent congestion too early will have a large effect if the current CWND is large, since it requires the CWND to be reset to the initial value again. The method that declares persistent congestion can still change, so we will not focus on this subject for the rest of this thesis.

[34] `https://github.com/quicwg/base-drafts/issues/3875`

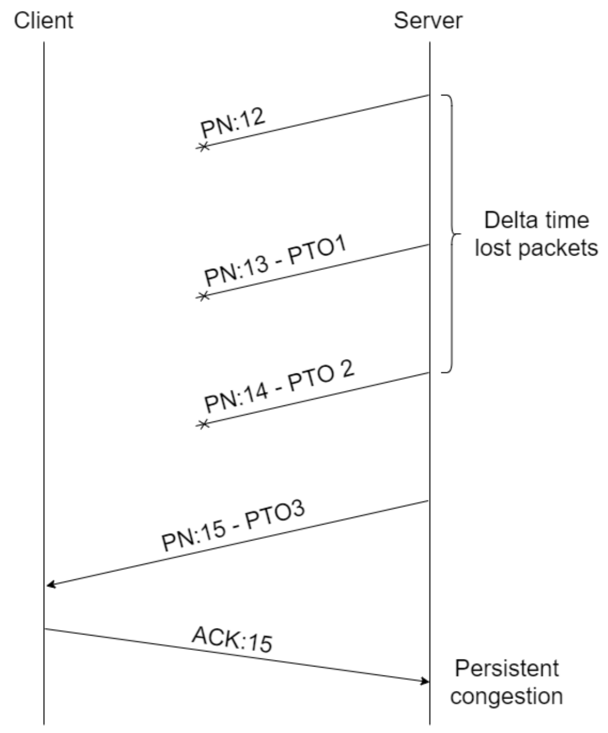


Figure 37: Example when persistent congestion is declared

6 Conclusion and Future Work

In this thesis we looked at the new congestion control and loss recovery logic that was being developed for the new transport protocol QUIC. The standardisation process for it has been ongoing for four years now, where the documents have been sent to the IESG in July 2020. Is the new congestion control and loss recovery logic ready to be deployed?

6.1 New protocol, new congestion control?

Even with QUIC being a new protocol, it would take over many of the different concepts TCP has, to provide reliable and efficient transport. When looking at the protocol structure in Section 2, we noticed there were some changes compared to TCP which are important for congestion control.

However, these changes are not radically different from what we see in TCP. Has there been a missed opportunity to complete revisit congestion control and redesign the acknowledgements, to provide a lot more information for congestion control? Of course, the challenge is that QUIC packets are almost fully encrypted, providing little information to the nodes in a network. ECN is still available and integrated into QUIC, but there has not been much research done on using ECN within QUIC.

6.2 Evolution in congestion control

When looking back at how congestion control has evolved over time from the early days of TCP (Section 3), we did notice a pattern emerging. Most of the time, a new congestion control algorithm would take over most parts from an older congestion controller, with only changing a small amount. Those changes were due to problems discovered in the older algorithm and fixes those.

There are not that many widely different concepts when it comes to congestion control. Most of the older the algorithms will focus on a particular signal for congestion, use a CWND to manage the send rate and recover lost data. First there was loss as a signal, not long after came delay as a signal for congestion. It is only recently we saw a completely different approach with BBR and COPA, where concepts as pacing were integrated for example.

The question then becomes why we are still discovering new things for congestion control after it has been used already for over thirty years. When looking for sources of the different congestion controllers, where issues were mentioned of older congestion controllers, how come that these issues were not immediately discovered when designing these new algorithms.

6.3 How to evaluate congestion control

During the process of creating a framework to test the QUIC and TCP congestion controllers (Section 4), we quickly discovered that this was not going to be simple. Even though testing over internet would provide us an interesting amount of data, in a real network environment. We would have no control over the state of the network, to which congestion controllers react to. Using a network simulator was then the other option, if this is a good option is difficult to answer.

After using the two simulators and analysing the results, we discovered that network simulators are not alike. This complicates the evaluation of congestion controllers, where the behaviour of the simulator causes the congestion controller to behave differently. Previous research that was done using only one simulator, are those results usable and dependant on the congestion controller, not the network simulator?

Using the knowledge about the simulators we were able to continue and evaluate the new congestion control and loss recovery logic in QUIC Section 5. When diving into the code in Linux and comparing it to the recovery draft, the differences discovered were on a smaller scale than expected. Seeing fixed and adaptive thresholds being used, using two ack-based loss detection methods instead of one, using a CWND counting in # bytes rather than packets... .

These difference that we were able to discover, did provide a much larger impact on performance then first expected. Even differences in implementations, such as sending an extra packet to fully utilise the CWND, can have a large impact on the generated throughput. Tweaking the fixed thresholds in QUIC or disabling an ack-based loss detection method, would improve or reduce performance by a large margin.

6.4 Future work

The results of the testing we have performed, showed that congestion control in QUIC is not finished. QUIC has the advantage of being in user space and thus allowing fast updates, without the need to upgrade the kernel. QUIC is a great platform for extensive testing, which will be needed to improve the recovery draft for QUIC. This document is available for developers who do not have much experience with congestion, which is understandable due to congestion control being a difficult subject. So, its first version should be a good baseline, that does show an improvement in performance compared to TCP.

Related to that, is to find a good process to evaluate congestion controllers and their performance. Network simulators are available but might not provide a simulation that is close to reality. Issues are better discovered early on so it can quickly be fixed, to prevent large scale issues all over the internet. With the use of eBPF and qlog, a lot more information is available with regards to congestion control. Is it possible to perhaps even automate the evaluation of congestion controllers, decreasing the time needed compared to doing it manually?

References

[1] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-29. Work in Progress. Internet Engineering Task Force, June 2020. 187 pp. URL: `https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-29`.

[2] Sally Floyd et al. *TCP Selective Acknowledgment Options*. RFC 2018. Oct. 1996. DOI: `10.17487/RFC2018`. URL: `https://rfc-editor.org/rfc/rfc2018.txt`.

[3] Sally Floyd, Dr. K. K. Ramakrishnan, and David L. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. Sept. 2001. DOI: `10.17487/RFC3168`. URL: `https://rfc-editor.org/rfc/rfc3168.txt`.

[4] V. Jacobson. "Congestion Avoidance and Control". In: *SIGCOMM Comput. Commun. Rev.* 18.4 (Aug. 1988), pp. 314–329. ISSN: 0146-4833. DOI: `10.1145/52325.52356`. URL: `https://doi.org/10.1145/52325.52356`.

[5] Dr. Vern Paxson and Mark Allman. *Computing TCP's Retransmission Timer*. RFC 2988. Nov. 2000. DOI: `10.17487/RFC2988`. URL: `https://rfc-editor.org/rfc/rfc2988.txt`.

[6] Kevin Fall and Sally Floyd. "Simulation-based comparisons of Tahoe, Reno and SACK TCP". In: *ACM SIGCOMM Computer Communication Review* 26.3 (1996), pp. 5–21.

[7] Lisong Xu, K. Harfoush, and Injong Rhee. "Binary increase congestion control (BIC) for fast long-distance networks". In: *IEEE INFOCOM 2004*. Vol. 4. 2004, 2514–2524 vol.4.

[8] Sangtae Ha, Injong Rhee, and Lisong Xu. "CUBIC: a new TCP-friendly high-speed TCP variant". In: *ACM SIGOPS operating systems review* 42.5 (2008), pp. 64–74.

[9] Robert L. Carter and Mark Crovella. "Measuring Bottleneck Link Speed in Packet-Switched Networks". In: *Perform. Evaluation* 27/28 (1996), pp. 297–318.

[10] Marek Małowidzki. "Simulation-based study of ECN performance in RED networks". In: (Jan. 2003).

[11] Mohammad Alizadeh et al. "Data Center TCP (DCTCP)". In: *SIGCOMM Comput. Commun. Rev.* 40.4 (Aug. 2010), pp. 63–74. ISSN: 0146-4833. DOI: `10.1145/1851275.1851192`. URL: `https://doi.org/10.1145/1851275.1851192`.

[12] Belma Turkovic, Fernando A Kuipers, and Steve Uhlig. "Fifty shades of congestion control: A performance and interactions evaluation". In: *arXiv preprint arXiv:1903.03852* (2019).

[13] Neal Cardwell et al. "BBR: Congestion-Based Congestion Control". In: *Commun. ACM* 60.2 (Jan. 2017), pp. 58–66. ISSN: 0001-0782. DOI: `10.1145/3009824`. URL: `https://doi.org/10.1145/3009824`.

[14] Mohd Mohamad, Mudassar Ahmad, and Md Ngadi. "Experimental evaluation of TCP congestion contorl mechanisms in short and long distance networks". In: *Journal of Theoretical and Applied Information Technology* 71 (Jan. 2015), pp. 153–166.

[15] Rasool Al-Saadi et al. "A Survey of Delay-Based and Hybrid TCP Congestion Control Algorithms". In: *IEEE Communications Surveys Tutorials* PP (Mar. 2019), pp. 1–1. DOI: 10.1109/COMST.2019.2904994.

[16] Jeonghoon Mo et al. "Analysis and Comparison of TCP Reno and Vegas". In: *Proceedings - IEEE INFOCOM* 3 (Jan. 2002).

[17] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. "TCP Vegas: New Techniques for Congestion Detection and Avoidance". In: *SIGCOMM Comput. Commun. Rev.* 24.4 (Oct. 1994), pp. 24–35. ISSN: 0146-4833. DOI: 10.1145/190809.190317. URL: https://doi.org/10.1145/190809.190317.

[18] Sumitha Bhandarkar et al. "Emulating AQM from end hosts". In: vol. 37. Oct. 2007, pp. 349–360. DOI: 10.1145/1282427.1282420.

[19] David Hayes and Grenville Armitage. "Revisiting TCP Congestion Control Using Delay Gradients". In: vol. 6641. May 2011, pp. 328–341. DOI: 10.1007/978-3-642-20798-3_25.

[20] K. Tan et al. "A Compound TCP Approach for High-Speed and Long Distance Networks". In: *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications.* 2006, pp. 1–12.

[21] Nandita Dukkipati et al. *Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses.* Internet-Draft draft-dukkipati-tcpm-tcp-loss-probe-01. Work in Progress. Internet Engineering Task Force, Feb. 2013. 20 pp. URL: https://datatracker.ietf.org/doc/html/draft-dukkipati-tcpm-tcp-loss-probe-01.

[22] Yuchung Cheng et al. *RACK-TLP: a time-based efficient loss detection for TCP.* Internet-Draft draft-ietf-tcpm-rack-09. Work in Progress. Internet Engineering Task Force, July 2020. 31 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-tcpm-rack-09.

[23] Venkat Arun and Hari Balakrishnan. "Copa: Practical Delay-Based Congestion Control for the Internet". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18).* Renton, WA: USENIX Association, Apr. 2018, pp. 329–342. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/arun.

[24] Jana Iyengar and Ian Swett. *QUIC Loss Detection and Congestion Control.* Internet-Draft draft-ietf-quic-recovery-29. Work in Progress. Internet Engineering Task Force, June 2020. 46 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-recovery-29.

[25] Quentin De Coninck. "Flexible multipath transport protocols". PhD thesis. UCL-Université Catholique de Louvain, 2020. URL: http://hdl.handle.net/2078.1/232084.

[26] Mark Allman. *TCP Congestion Control with Appropriate Byte Counting (ABC)*. RFC 3465. Feb. 2003. DOI: 10.17487/RFC3465. URL: https://rfc-editor.org/rfc/rfc3465.txt.

[27] Josh Blanton et al. *Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)*. RFC 5827. Apr. 2010. DOI: 10.17487/RFC5827. URL: https://rfc-editor.org/rfc/rfc5827.txt.

[28] Matthew Mathis and Jamshid Mahdavi. “Forward Acknowledgement: Refining TCP Congestion Control”. In: *SIGCOMM Comput. Commun. Rev.* 26.4 (Aug. 1996), pp. 281–291. ISSN: 0146-4833. DOI: 10.1145/248157.248181. URL: https://doi.org/10.1145/248157.248181.

[29] Matt Sargent et al. *Computing TCP’s Retransmission Timer*. RFC 6298. June 2011. DOI: 10.17487/RFC6298. URL: https://rfc-editor.org/rfc/rfc6298.txt.