Tor over QUIC

Jaap Heijligers

Abstract

Tor is the most popular tool for anonymous online communication. However, the performance of Tor's volunteer-run network is suboptimal when network congestion occurs. Within Tor, many connections are multiplexed over a single TCP connection between relays, which causes a head-of-line blocking problem, degrading relay performance. In this thesis, Tor's TCP transport layer protocol is replaced by QUIC, a UDP-based protocol that natively supports multiplexing streams asynchronously, effectively solving head-of-line blocking. Its performance is evaluated within various network environments through Containernet, a flexible Docker-based network test bed that allows for simple reproduction of results. Along with testing multiple congestion control algorithms, the impact of using Hystart++ within Tor over QUIC is evaluated. It is found that QUIC over Tor can perform up to 50% better in time to last byte performance than vanilla Tor in a realistic network environment, while featuring more consistent time to first byte performance. Additionally, the evaluations shows that throughput consistency and fairness amongst downloaders are improved as well, Besides offering improved performance, Tor over QUIC is designed with deployability and security in mind. This makes QUIC an attractive replacement as Tor's transport layer protol.

Acknowledgements

Thanks to Stefanie, who has been a great supervisor. She consistently provided valuable feedback and guidance in our weekly meetings. Thanks to Tasha, who is now my fiancée, for always supporting me and for proofreading the thesis. Thanks for Wladimir and Cyril for helping with technical aspects of the thesis, and furthermore, thanks to my friends and family for being supportive throughout.

Contents

List of Tables

List of Figures

1. Introduction

As tracking, surveillance and censorship become more widespread, methods to increase privacy on the Internet become increasingly important. One way to increase online privacy is by using Tor[1]. Tor is free and open source software that lets its users connect to the Internet anonymously. It does this by directing the user's internet traffic through the Tor network: a set of volunteer-run Tor servers that conceal all traffic that goes through them. Using Tor makes it difficult for adversaries to obtain the user's location. Additionally, the user's online behavior becomes difficult to track when they use Tor. Amongst its many users, whistleblowers and journalists use Tor to protect themselves from corrupt governments[2].

The Tor network has gained popularity over time, but its performance is limited by the amount of volunteer-run relay servers. Currently, the Tor network consists of over 7000 relays[3], that in total serve more than 2 million users. The packets that a user sends are routed through a Tor circuit, a path through 3 relays, each located in a different country, before arriving at the destination server. When the Tor network has too many users and not enough relays to handle their traffic, the network becomes overburdened and network performance suffers.

Because there are many more users than relays, and each user's circuit passes through 3 relays, the Tor network can be seen as many interconnected relays. Any relay is connected to many relays, and any two relays are part of the circuit of many users.

To reduce resource consumption, all circuits between two relays are multiplexed over a single TCP connection rather than using a single TCP connection for each circuit. However, TCP ensures in-order delivery, which means that if a packet of one circuit is dropped, all other circuits between these two relays are on hold until the packet is retransmitted. This problem, known as head-of-line blocking, limits the efficiency of the Tor network, especially as more packet loss occurs.

QUIC[4] is a UDP-based transport layer protocol which can be used as a replacement for TCP. QUIC features various improvements to performance, latency and congestion control. Additionally, QUIC features native multiplexing by supporting multiple individual streams within one QUIC connection. All streams within a QUIC connection operate independently, and each stream has its own congestion and flow control. By using QUIC, when a packet is dropped, only the streams that correspond to this packet are on hold while the packet is resent, while other streams can continue transmission. This is ideal for applications which rely heavily on multiplexing, such as Tor.

Various research has been done on the performance of Tor. Jansen et al. propose improvements to Tor's scheduler and Tschorsch and Scheuermann[5] show that using a backpressure-based flow control algorithm can improve performance and fairness in the Tor network. However, these approaches do not eliminate head-of-line blocking, leaving a need for an alternative transport layer protocol. Various alternative transport layer protocols have been proposed as well. Viecco[6] proposes using TCP across the entire circuit, rather than between each two relays. Reardon and Goldberg[7] propose using a user-space TCP implementation on top of DTLS, and Nowlan et al.[8] aim to eliminate head-of-line blocking by circumventing TCP's in-order delivery through a kernel patch. However, the deployability of these approaches is severely limited. Ossification prevents the deployment of many unpopular transport layer protocols, and the requirement of kernel modifications makes deployment of Tor infeasible. QUIC has been proposed for integration with Tor as well, by both Clark[9] and Sabée[10]. However, the former uses a largely unmaintained library, and the latter implementation proved unstable while its source code is unavailable for improvement. Additionally, these works use an Internet Draft version of the QUIC protocol while QUIC is now a Proposed Standard, and only limited experimentation is conducted within these papers.

In this thesis, QUIC is implemented in Tor as an alternative to the current TCP-based connection between Tor relays. Because the QUIC protocol has only recently been promoted to a Proposed Standard, rather than an Internet draft, this is, to our best belief, the first work that integrates version 1 of the QUIC protocol. The Quiche library is used, which now implements QUIC version 1. It is backed by Cloudflare, and the library is used in production throughout Cloudflare's network.

Integrating QUIC with Tor requires several design decisions to be made. It is decided to implement QUIC hop-by-hop rather than end-to-end. This approach can take full advantage of QUIC's multiplexing and comes with less complexity, while an end-to-end approach comes with the risk of compromising anonymity and is difficult to deploy.

Protecting user privacy is the core of Tor's application. Because a hop-by-hop design is used, the impact of QUIC on security and privacy is limited. Due to the use of header encryption, QUIC exposes less information than TCP, which makes attacks on anonymity more difficult. Even though more research on the anonymity implications needs to be done, this inspires confidence that the integration of QUIC does not degrade Tor's protection of user privacy.

Tor currently only supports TCP as its transport layer protocol. Since QUIC is UDP-based, and UDP is a connectionless protocol, core parts of Tor had to be redesigned. Additionally, to use QUIC's multiplexing effectively within Tor, a major refactor of Tor's connection and scheduling design was required. To allow Tor to fall back to TCP over TLS at runtime when QUIC is unsupported, QUIC has been implemented independently of Tor's current transport layer protocol, such that both designs can be used interchangeably. To allow future research to base their work on the implementation efforts of this thesis, the source code is made available publicly.

Both the QUIC protocol and the Quiche library are ready for production use,

no kernel patch is required, and maintenance of the QUIC library is done by a large company. QUIC, through Quiche, comes closer to a feasible alternative for Tor's transport layer protocol than previous proposals.

The implementation of QUIC, which is referred to as Tor over QUIC, is evaluated through Containernet. Containernet[11] is a network emulator that uses Docker containers as its hosts. It allows for defining network topologies with restrictions imposed to its links, such as limited latency and added loss. Additionally, the use of Docker makes experiments that use it more robust and reproducible.

Beyond finding the difference between QUIC and vanilla Tor, the impact of several of QUIC's congestion control configurations are explored. QUIC's performance is evaluated while using the CUBIC and Reno congestion control algorithms. Additionally, the impact of enabling Hystart++, an improvement of the slow start mechanism in congestion control, is explored.

The impact of latency, loss and jitter on Tor are evaluated, as well as realistic network conditions, fairness between users, and the performance of the Tor network when it is under heavy load.

It is shown that QUIC performs better when using CUBIC than when using Reno in every scenario. Enabling Hystart++ is shown to always be beneficial when using CUBIC. However, when using Reno, Hystart++ can lead to performance degradation in a loaded network.

Overall, QUIC with CUBIC and Hystart++ shows better Time to Last Byte performance than vanilla Tor in each evaluated scenario. In a loaded Tor network, QUIC performs 50% better than vanilla Tor. QUIC's advantage increases as more loss is added to the network environment. Additionally, QUIC is shown to provide more fairness to its users than vanilla Tor. When two users download through the network simultaneously, the available bandwidth is not only higher, but also distributed more equally amongst them. Overall, QUIC is shown to be a very promising alternative to Tor's transport layer protocol, in terms of performance, fairness and deployability. The code of the Tor adaptations, and of the experimental test bed are open source such that results can be reproduced easily.

Overview of Thesis

In chapter 2, background on both transport layer protocols and Tor is provided, with focus on performance. In chapter 3, previous work is discussed in detail. The design goals, implementation of QUIC within Tor and an overview of the network test bed are laid out in chapter 4. Chapter 5 includes a performance evaluation, in which Tor over QUIC's performance is explored in relation to vanilla Tor. Chapter 6 discusses future work and chapter 7 concludes this thesis.

2. Background

In this thesis, the transport protocol that is used in the Tor network is compared to QUIC in relation to performance. This chapter provides background on transport protocols along with their performance characteristics. Then background on Tor is provided, along with Tor's performance model.

2.1 Transport layer

Communication over the Internet is a complex problem that has been researched extensively over the past decades. This problem can be approached by dividing intercommunication up into layers, each with their own sub-problems. Most commonly, the standardised OSI model is used to model computer system communications.

The OSI model divides the responsibilities of communication into 7 layers. Whereas the higher layers are more application and use-case specific, the lower layers are more general and application independent.

Table 1: Layers with their functions within the OSI model

#	Layer	Function
7	Application	High-level APIs
6	Presentation	Encoding, compression, encryption
5	Session	Communication sessions
4	Transport	Reliable transmission, segmentation, acknowledgement, multiplexing
3	Network	Multi-node networking, addressing, routing, traffic control
2	Data link	Reliable transmission of data through a physical layer
1	Physical	Transmission of raw bit streams over a physical medium

The bottom two layers ensure that bit streams can be sent reliably over physical mediums between devices, such as Wi-Fi and Ethernet. The network layer provides addressing within a network of devices. For example, IP, the Internet protocol, is a network layer protocol that specifies how packets can be routed

throughout the Internet. The transport layer builds on top of the network layer and describes end-to-end error control. The most common transport layer protocols are TCP and UDP. On the Internet, these protocols are used on top of IP.

User Datagram Protocol (UDP)

UDP is a relatively simple transport layer protocol, it is used to send messages, called datagrams, from one host to another. UDP is used on top of IP and provides checksums for limited data integrity checking. UDP is lossy, which means that when a datagram is dropped, the protocol does not ensure that the datagram is resent and delivered to its destination. UDP does not ensure that its datagrams arrive in the order they are sent either. Due to its simplicity and low overhead, UDP is often used in time sensitive applications, such as Internet calls and video streams.

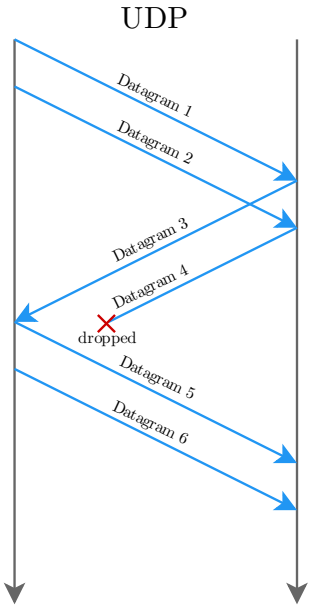

Figure 1: UDP datagrams can get lost or delayed, and UDP does not detect or retransmit these. Effectively, this makes UDP a lossy protocol.

Transmission Control Protocol (TCP)

TCP is a transport layer protocol that, contrary to UDP, ensures reliable and in-order delivery. Where UDP is connectionless, in TCP a connection is established before data can be sent over it. To establish a TCP connection, a 3-way handshake is done, as seen in figure 2. After successful connection establishment, the TCP connection is considered open. The connection can now transport a stream of bytes.

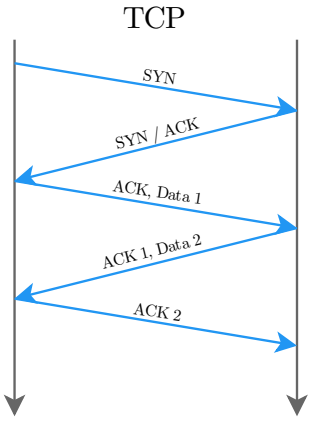

Figure 2: TCP connection establishment. After the 3-way handshake (SYN, SYN/ACK, ACK), data can be sent. Each received segment is acknowledged such that missing segments are detected and retransmitted. TCP provides a reliable in-order bytestream to its upper layer.

In UDP, messages are called datagrams. TCP's messages are called segments. In TCP, each sent segment is acknowledged by the receiver. If the sender notices that it did not receive an acknowledgement within some time, it will consider the packet lost, and the packet will be retransmitted. Additionally, packets come with a sequence number. The sequence number represents the byte number of the first byte of the packet within the stream. If the receiver receives a packet with a sequence number that doesn't match the amount of received bytes, i.e., a gap in the bytestream occurs, a lost packet is detected. In this case the receiver will not expose the new data to the upper layer until the gap is recovered. This ensures that packets are delivered to the session layer reliably and in order. An example of retransmission after packet loss can be seen in figure 3.

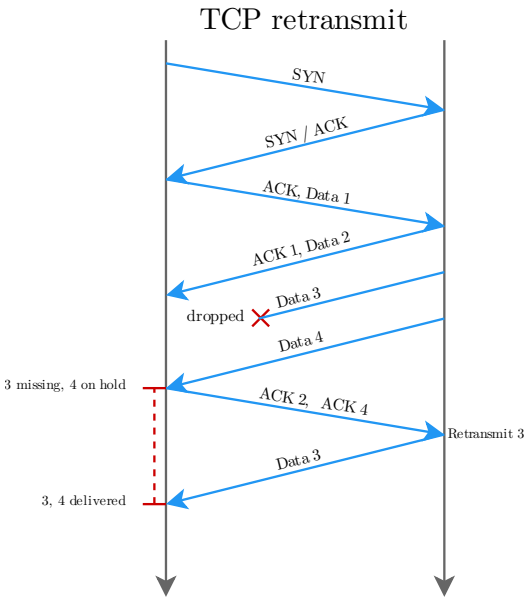

Figure 3: TCP retransmission. The left side receives segments with Data 2 and 4, but not 3. Acknowledgements are sent for the received segments. When the right side notices that the segment with data 3 was never received, it is retransmitted. Data 4 is on hold on the left side until the retransmission of 3 is received, such that all segments are delivered in-order to the upper layer.

TCP uses flow control, through which the receiving end can specify how much more data can be sent by the receiver before waiting for an acknowledgement. This prevents the receiver from getting overwhelmed when the sender can send packets faster than the receiver can process. In TCP, this is implemented as a sliding window. Through the sliding window, the receiver tells the sender how many unacknowledged bytes can be sent. If the sender sends bytes the size of the full window, it has to wait until some packets are acknowledged before it is allowed to send any more bytes. This prevents the receiver from receiving more data than it can handle.

Additionally, congestion control is used in TCP to detect when the network, rather than the receiver, is overburdened. For example, congestion can be detected through packet loss, i.e. when no acknowledgment is received for the packets that it has sent. When this is the case, the sender knows that some packets are lost or delayed, likely due to network congestion. As a result, the sender will lower its sending rate based on the used congestion control algorithm. This prevents overburdening the network.

Congestion control algorithms

Within TCP, congestion control is based on a congestion window. This window denotes the maximum amount of unacknowledged bytes that can be in flight at any point. A congestion control algorithm is in charge of growing this window

when no congestion is detected i.e., the network can handle higher bandwidth. Additionally, when congestion is detected, the congestion control algorithm decides how much the congestion window is lowered. Two popular congestion control algorithms are Reno and CUBIC.

Fast retransmit

Normally, TCP detects loss by waiting for an acknowledgement. An acknowledgement is expected after roughly one round trip time (RTT), so if a packet isn't received after a small multiple of the measured RTT, loss is detected. Because this delay in loss detection limits performance, an alternative method of detecting loss was introduced, which is called fast retransmit. If the receiver receives packets out-of-order, e.g. if after packet #1 and #2, packet #4 is received, the receiver resends the acknowledgement for packet #2, which is still the last in-order packet it has received. The sender then receives a duplicate acknowledgement for packet #2. If the receiver now receives packet #5 and packet #3 is still not available, a third acknowledgement of packet #2 is sent.

If the sender receives two acknowledgements for the same packet, this could either mean that packet re-ordering has happened, or a packet has been lost. However, when a third duplicate acknowledgement has happened, then the sender can be reasonably certain that the packet was lost and a fast retransmit will occur. This allows the sender to retransmit a packet before the acknowledgement timeout for the packet occurs. As such, a packet is retransmitted faster than it would be by only relying on acknowledgement timeouts.

Reno

Reno is a basic congestion control algorithm for TCP. It uses three phases: slow start, fast recovery and congestion avoidance.

It starts within the slow start phase. In this phase, the congestion window is roughly doubled every RTT. This continues until either packet loss is detected, which is followed by fast recovery, or the congestion window has reached a certain threshold (sstresh), which is followed by congestion avoidance.

If three duplicate acknowledgements are received, Reno enters its fast recovery state. In this state, the congestion window is halved. Reno stays in fast recovery until it has recovered from the packet loss. After that, congestion avoidance is entered.

In congestion avoidance, Reno roughly adds the size of one packet to the congestion window for every RTT. It continues to do so until packet loss is detected, and fast recovery is entered.

If an acknowledgement timeout occurs, rather than receiving three duplicate acknowledgements, the congestion window is reset to its initial value, and the slow start phase is entered.

CUBIC

CUBIC's main difference from Reno is in its congestion avoidance phase. Instead of increasing the congestion window linearly, it features concave growth towards

the last window size at which loss occurred. This ensures that the congestion window is maximized in case loss happens at this same window size again. Then, convex growth is used to grow beyond this point. This is done to make sure that if the network allows for higher bandwidth now, that it is found swiftly. The growth function, which is based on the last window size at which loss occurred, can be seen in figure 4.

Additionally, when loss occurs, the congestion window is only reduced by 30% in CUBIC rather than 50% in Reno.

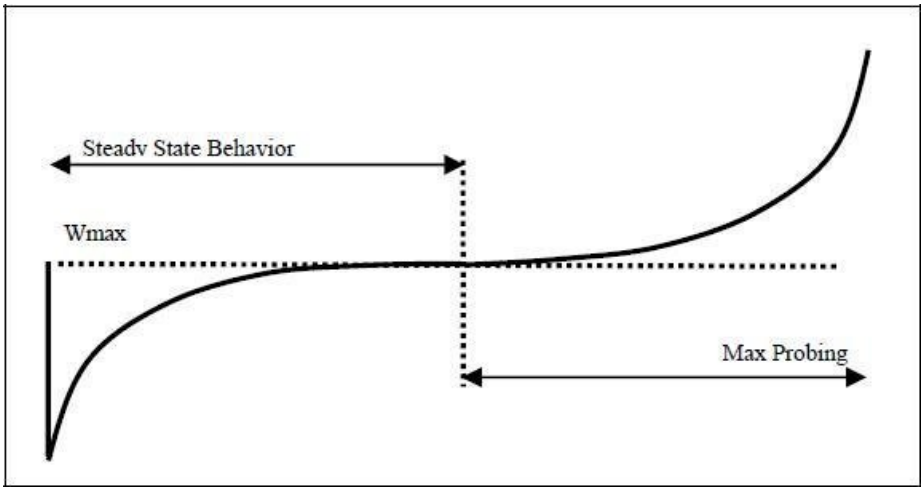

Figure 4: Congestion window growth in CUBIC's congestion avoidance phase[12]

Hystart

Slow start tries to find a bandwidth limit swiftly by doubling the congestion window every RTT. Setting the slow start threshold too low will start congestion avoidance too soon which risks using a too low congestion window. Setting the slow start threshold too high will result in loss and leads to immediate reduction of the congestion window.

To find a middle ground, Hystart is proposed, which is created by the same people behind CUBIC. Later, Hystart was improved upon by Microsoft engineers in the form of Hystart++, which is currently an IETF Internet Draft[13] [14].

Within the slow start phase, Hystart detects an increase in RTT, and attempts to exit slow start before actual packet loss occurs. This helps to converge to an optimal congestion window quicker, as slow start's exponential congestion window growth can lead to overshooting the optimal congestion window. Additionally, Hystart++ adds a limited slow start (LSS) phase between slow start and the congestion avoidance phase. This is done because RTT fluctuations often happen before the optimal congestion window is found. LSS will grow the congestion window faster than congestion avoidance, but much slower than slow start's exponential growth. LSS is used until packet loss is detected, after which fast recovery, and then congestion avoidance are used.

Multiplexing

There is a limit to how many TCP connections a server can handle. Additionally, if a client create multiple TCP connections to the same server, the three-way-handshake has to be done for each connection. To mitigate this extra latency, and to keep servers from being overburdened, multiplexing can used to send multiple streams of data over the same TCP connection.

A typical use case for TCP multiplexing is web browsing. A web browser requests multiple resources from a server through HTTP. In HTTP/1.0, an early version of HTTP, one TCP connection per HTTP request was made. To improve performance, HTTP/1.1 defaults to keeping TCP connections open such that multiple HTTP requests can be done in succession over the same connection. This only requires one TCP handshake to be done. Additionally, HTTP/1.1 supports pipelining: multiple HTTP requests can be sent over the connection before receiving any responses, rather than doing each request-response successively. The requirement of returning responses fully, in the same order as the corresponding requests, is a limitation of HTTP pipelining.

HTTP/2 improves on this model by adding streams on top of the TCP connection. Each HTTP request is sent over a separate stream such that responses can be sent in arbitrary order. Additionally, HTTP/2 allows responses to be sent in chunks, which allows responses to be intermingled.

Due to HTTP/2's reliance on TCP, a design mismatch occurs. Where HTTP/2 sees its connection as a collection of separate streams, TCP still operates as a single byte stream. When a packet is dropped, TCP's in-order-delivery property requires it to hold on to any consecutive packets until the dropped packet is retransmitted. This means that when a packet for one HTTP/2 stream is dropped, the application cannot access arriving packets from any stream until the TCP connection is recovered. This delay is called head-of-line blocking [3].

Tor's design is similar to HTTP/2. Tor multiplexes separate circuits over a single TCP connection. Similarly, Tor also suffers from head-of-line blocking. When a packet is dropped, the receiver does not have access to incoming circuit packets until the lost packet is retransmitted.

TLS

While the transport layer is responsible for packets reaching their destination in a controlled manner, it does not attempt to protect this data from potential adversaries. Encryption can be used to deny man-in-the middle attacks, and to guarantee users' privacy. Notably, TLS is a protocol that runs on top of transport layer protocols, and provides authentication and encryption.

To create a TLS connection, a handshake is done in which certificates are exchanged and verified. After this is done and the client accepts the server's identity, a cipher is decided upon. This cipher is then used to encrypt the underlying byte stream according to their exchanged shared secret.

Due to TLS's requirement for a reliable transport, TLS is most often used on top of TCP. However, adaptions have been made to make TLS work with the unreliable UDP protocol, in the form of DTLS.

TCP only allows higher layers to exchange data after its three-way handshake has been completed. When TCP is used in combination with TLS, their combined handshakes require three full round trips between hosts to set up the connection with encryption. This imposes a latency cost on all new connections.

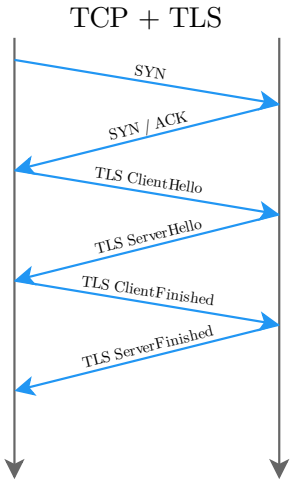

Figure 5: TCP + TLS handshake. Because of separation of concerns between layers in the OSI model, the TLS handshake can only start after 1 RTT of the TCP handshake. Because the TLS handshake takes up 2 RTT, it takes 3 RTT before application data can be sent.

QUIC

QUIC is a protocol built on top of UDP that addresses multiple of TCP's problems in modern use cases. It has built-in support for multiplexing, and uses TLS 1.3 internally to provide encryption. Furthermore, QUIC aims to reduce latency and congestion.

While QUIC is used on top of UDP, it implements many of the features that TCP provides such as reliability and in-order delivery. Additionally, QUIC replaces TLS, which is often used on top of TCP, by including the TLS 1.3 handshake in its connection establishment. As such, QUIC doesn't fit in one single layer in the OSI model. Instead, it takes on the responsibilities of multiple layers and combines those efficiently. QUIC's position within the OSI model is illustrated in figure 6.

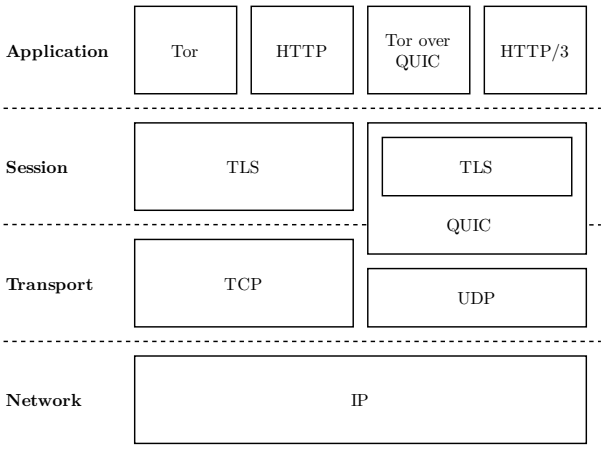

Figure 6: QUIC within the OSI model. While QUIC is implemented on top of UDP, it implements some of the features of UDP's sibling, TCP, such as reliability, in-order delivery, and congestion control. While TLS can run on top of TCP independently, QUIC embeds TLS in its handshake. This allows QUIC to cut down on handshake latency.

Handshakes

In TCP, after sending a SYN packet, a client has to wait for an acknowledgement before it can send any higher layer data. After this TCP handshake, TLS can start its own handshake, in the form of supported protocols and a key exchange. Because QUIC replaces both TCP and TLS, both handshakes are merged into one.

TLS 1.2 and earlier require two full round trips for their handshake. TLS 1.3[15] improves on this by combining the exchanges of supported ciphers and the exchange of keys. Where a client and a server would first agree on which cipher to use before exchanging cryptographic keys, the client guesses a supported cipher suite, and immediately sends its key share along.

Additionally, TLS 1.3 introduces zero round trip time (RTT) resumption. Through this, the server sends a pre-shared key (PSK) to the client on initial connection. Whenever the client reconnects to the server, it can send along this PSK to authenticate. Encrypted application data can now be sent immediately, without needing to wait for a reply. A drawback of this method is its lack of protection against replay attacks. Even though the data of the resumption packet is encrypted, an adversary could record the data and resend it to the server. In this case, TLS provides no mechanism for the server to protect against this.

Since QUIC comes with TLS 1.3 support, and combines the transport layer and TLS handshakes, latency can be reduced dramatically over a TCP + TLS setup. Where the latency of a traditional reliable connection is 3 round trip times, QUIC brings this down to 1 RTT for initial connections. Additionally, this can be brought down to 0 RTT for follow-up connections, though extra care has to be taken to protect against replay attacks. An overview of the QUIC

handshakes can be seen in figure 7.

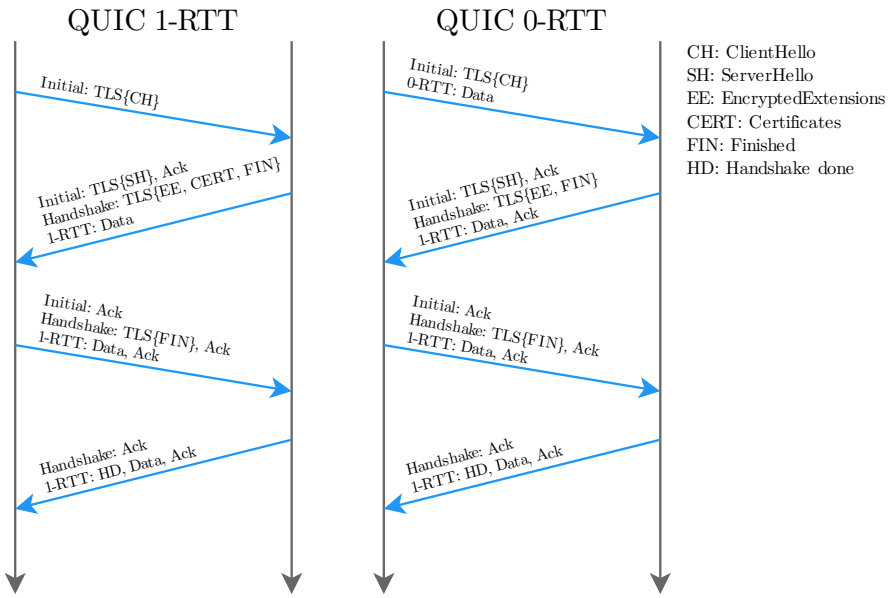

Figure 7: QUIC Handshakes. QUIC uses a 1-RTT handshake for its first connection. In later connections to the same server, previously exchanged keys can be used for a 0-RTT handshake. Initial, Handshake, 0-RTT and 1-RTT are QUIC packet types. Multiple QUIC packets can be coalesced into a single UDP datagram[4]. In both scenario's, the server can immediately send data back, which is called "0.5-RTT data."

Streams

QUIC has multiplexing built-in through streams. Each QUIC connection can contain multiple streams, and each stream has its own priority, congestion control and flow control. Data within streams is delivered in-order, but streams are independent of each other. Whereas the TCP based HTTP/2 protocol provides a similar multiplexing mechanism, TCP's in-order delivery property can still cause head-of-line blocking. In HTTP/2, as soon as a stream packet is sent over TCP, it will be delivered in the order that it is passed to TCP, even if packets of other streams are blocked in the meantime. QUIC solves this problem by relying on UDP and implementing in-order delivery on a per-stream basis.

Head-of-line blocking can still occur in QUIC, but this will happen for a specific stream, as only individual streams guarantee in-order-delivery. Packet loss on one stream will not affect the performance of other streams within the same QUIC connection.

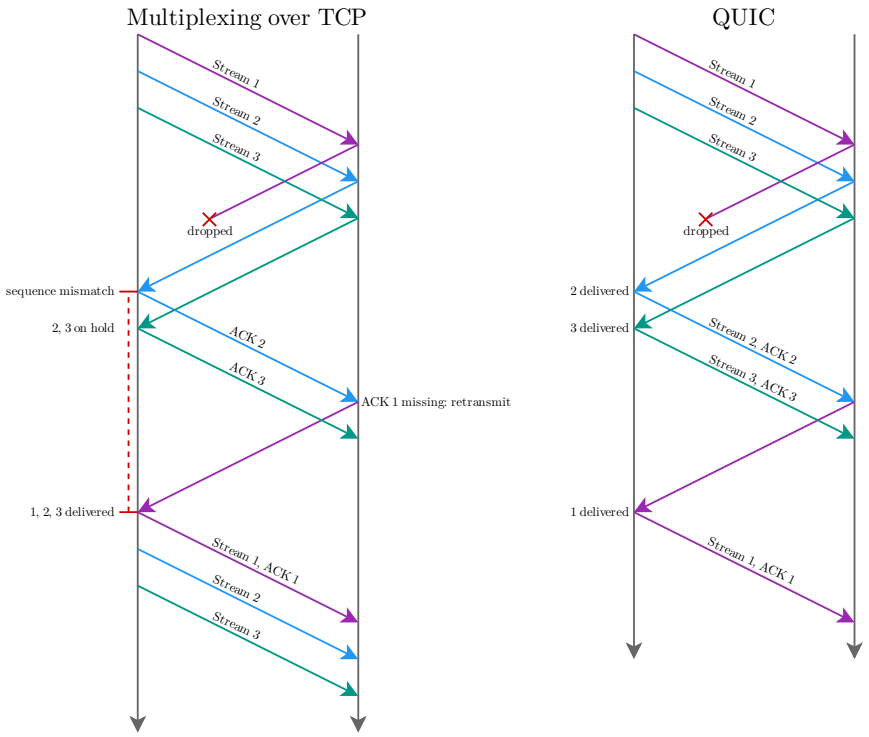

Figure 8: Multiplexing head-of-line blocking in TCP and QUIC. TCP ensures in-order delivery on the connection level where QUIC ensures this only per-stream. In TCP, when a segment is dropped, all other segments have to wait for its retransmission. In QUIC, a dropped packet affects only the packet's stream, and packets from other streams will still be delivered on arrival.

Ossification

The Internet does not only consist of client and servers, middleboxes such as NATs, firewalls, intrusion detection systems, and proxies are increasingly prevalent[16]. These middleboxes can look into the contents of packets such as protocol metadata. Some middleboxes will route or deny packets based on this information. Unfortunately, many middleboxes are implemented incorrectly. This has been a problem for rolling out both new protocols and new versions of existing protocols.

In 2017, TLS 1.3 was deployed to a part of Chrome and Firefox users. However, a larger connection failure rate than expected was perceived amongst this group. This was caused by middleboxes that were written for compatibility with older TLS versions, but failed to work well with this new version[17]. Eventually a workaround for this problem was used by advertising the TLS 1.2's version in TLS 1.3 packets while introducing a new versioning mechanism. In this case, the widespread usage of faulty middleboxes forced the protocol to adopt a workaround, adding complexity and sacrificing efficiency. In a similar fashion, it is found that TCP headers are used and even modified by middleboxes in

various ways, which makes the deployment of some TCP extensions difficult[16]. This reduction in flexibility in protocol design, which is caused by middleboxes, is called protocol ossification.

Protocol ossification makes deploying new protocols difficult as well. SCTP is a transport layer protocol that was developed in 2000. It supports multiplexing multiple streams over a single connection. However, middleboxes do not recognize the protocol and some simply block it. This has made it impossible to use the protocol reliably over the Internet. As a result, SCTP is mainly used in controlled environments[18].

A solution to ossification is use of encryption. If a middlebox cannot decrypt a packet, middleboxes cannot make decisions based on its contents. Middleboxes are still able to drop all UDP packets, however, the scope of damage they can do to the evolution of the protocol is severely limited. For this reason, QUIC encrypts not only application data, but also its own headers. This allows the QUIC protocol to evolve without the risk of middleboxes changing headers based on a specific version. Additionally, QUIC is already implemented in major web browsers and used on the web. This makes it unlikely that new middleboxes will block QUIC in its entirety.

One of the main goals of QUIC was to be deployable immediately, without waiting for middleboxes to update. This excluded any protocol but TCP and UDP from consideration, as other transport layer protocols are typically blocked or degraded by middleboxes[19]. Additionally, evolving TCP to circumvent in-order-delivery to prevent head-of-line blocking appeared infeasible. While modification of TCP is plausible, major modifications such as this could take 10 years to deploy. This is mainly because TCP is implemented in operating system kernels, which are generally slow to update. When based on UDP, most of QUIC can run in user space, which allows for faster protocol updates in the future.

QUIC version 1

Google initially designed QUIC internally, it published details about the protocol to the public in 2013[20]. In 2015, an Internet Draft of the specification was submitted to the IETF, and a QUIC working group was established. In May 2021, after iterating through draft versions, the IETF officially formalized QUIC as RFC 9000[4]. This means that QUIC version 1 is now a proposed standard, which may become an Internet Standard in the future.

Besides RFC 9000, the IETF published RFC 8999, 9001 and 9002 as well. These describe version-independent properties of QUIC, using TLS to secure QUIC, and loss detection and congestion control within QUIC, respectively.

Congestion and flow control

QUIC's congestion and flow control differ from TCP in a couple of ways, as specified in RFC 9002[21].

Packet numbering

QUIC packets use one of three types of encryption: Initial, Handshake, or Application data. QUIC packet numbers are tracked for each encryption level individually i.e., each encryption level comes with its own packet number space. For each connection, all QUIC packet numbers start at 0 and increase monotonically. Even when a QUIC packet is retransmitted, the packet number of the retransmitted packet is higher than the packet number of the original packet. This is different from TCP, which assigns the original sequence number to retransmitted segments. QUIC does not allow repeating packet numbers as to prevent ambiguity, it allows for endpoints to distinguish between original and retransmitted packets.

Acknowledgements

RFC 2018[22] introduces selective acknowledgements (SACKs) in TCP through which a receiver can selectively acknowledge segments that are received after a gap. Through these SACKs, the sender only has to retransmit the missing packets instead of all packets since the detected gap. QUIC comes with a similar mechanism. Where TCP can only specify three SACK ranges in a single packet, QUIC allows for many SACK ranges. This allows QUIC to recover quicker from loss in high loss, high throughput scenarios.

TCP allows a receiver to discard unacknowledged data, even if this data is part of a selective acknowledgement range. Discarding data that has been acknowledged through SACKs, but not yet by a normal acknowledgement, is called reneging. This can occur when the receiver runs out of buffer space. However, this adds complexity and memory pressure to the sender, as its sent data cannot be freed after a SACK has been received, only after a normal ACK. For this reason, QUIC does not allow reneging at all.

Additionally, QUIC contains improvements to timeout calculation, the minimum congestion window is increased to from one to two packets, and handshake packets are not handled differently than other packets anymore in QUIC.

2.2 Tor

Internet packets contain an origin and destination address. The origin IP address can be used to trace a packet back to its sender. Additionally, this IP address can be used to roughly determine the geographic location of the user who sent the packets[23]. Tor[1] is an overlay network that aims to protect its users against tracking, surveillance and censorship. This is done by encrypting the user's Internet traffic and routing it through multiple hops such that no direct connection between the user and its destination exists. This makes it difficult for an adversary to know where the user's traffic is coming from, and what kind of traffic the user is sending.

Tor's source code[24] and Tor's specification[25] live in separate source code repositories.

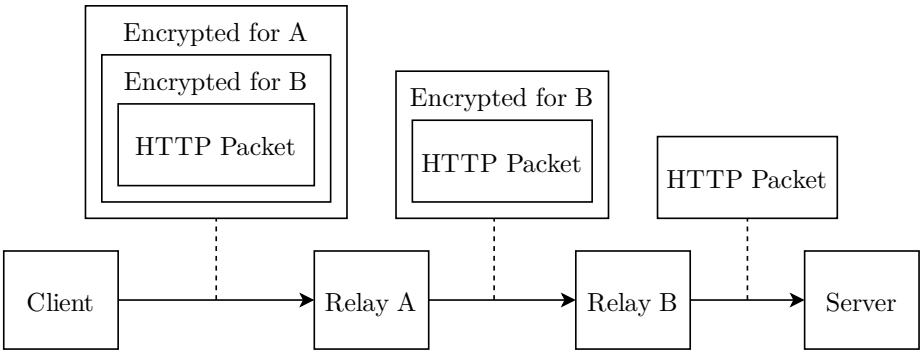

Figure 9: Tor Onion Routing. Cells are encrypted by the client in several "layers," one for each relay in the circuit. Each next relay "peels" a layer of encryption off, the circuit id becomes visible and the relay now knows to which relay the remaining data should be forwarded. The innermost "skin" is decrypted by the exit node, and sent to the destination server.

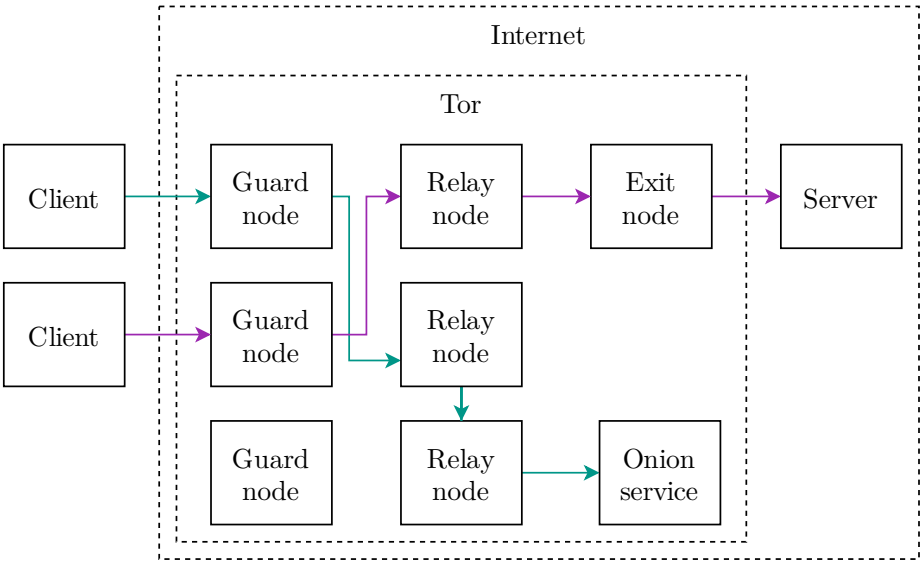

Figure 10: The Tor Network. By default, three hops are used between the client and the destination. The last hop must be an exit node for connections towards a server outside of the Tor network. However, onion services operate within the Tor network, this allows the server to operate anonymously too, rather than only the client.

Circuits

When a user connects to a server through Tor, Tor first tries to create a circuit. This is done by asking one of the dedicated directory authorities about available relays. Tor picks three random relays, proportional to the amount of bandwidth they can handle, and starts creating a path between them. First, it connects to the first relay, then it sends an extend request telling the first relay to connect to the second relay. This process is repeated for the second relay to connect to the third relay. When all relays accept this circuit, the circuit is opened, and the user can now send data through it.

When Tor is used to connect to a server on the Internet, the last relay in the Tor circuit connects to the destination server. This means that the server can see the IP address of the last relay in the circuit. Additionally, this last relay can see the data that the user wants to send to the destination server.

Because the last relay in the circuit bridges the Tor network with the rest of the Internet, it is called an exit node. Not all relays in the Tor network are exit nodes. As such, some relays can only be used at the start or in the middle of a circuit.

Streams

Tor functions as an overlay network that forwards TCP streams. When a user runs the Tor client, a SOCKS proxy is started. Any application that supports

SOCKS can use this to route its TCP streams through the Tor network. For example, the user can configure their browser to use the Tor client as its proxy. Now, the browser's HTTP traffic will be sent through a Tor circuit. Internally, Tor uses a stream identifier to distinguish between TCP streams within a circuit. These stream ids are end-to-end encrypted such that only the client and exit node can distinguish TCP streams. Other relays in the circuit will know to which circuit the data belongs, but not which stream.

Cells

All data that is sent over circuits is packed into cells. Cells contain an identifier for the circuit it belongs to and a command which specifies the purpose of the cell. Additionally, cells contain a payload, optionally with padding. There are two types of cells: fixed-width cells and variable-length cells. Fixed-width cells use padding such that each of the cells has the same length. This is done to protect against attacks that use the size of cells to infer what kind of data is sent. Tor nodes use fixed-width cells to forward end-to-end stream data. Relays use variable length cells for their initial handshake. These cells can contain certificates, supported versions or authentication data.

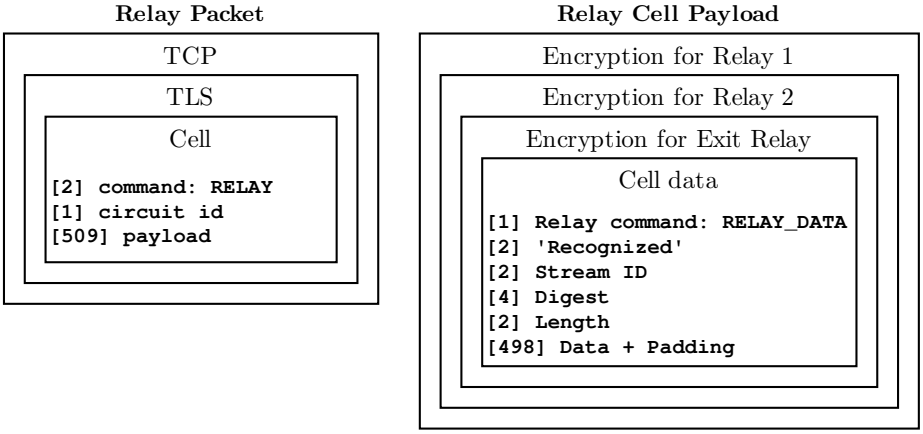

Figure 11: Relays exchange cells with "command" and "circuit_id" fields and a payload. The relay cell payload is encrypted for each individual relay on the path, these are the onion's "layers." Each relay "peels off" a layer by decrypting the payload through its private key. The exit node can then decrypt the cell data. A "RELAY_DATA" cell contains a stream id that corresponds to an end-to-end TCP connection and a payload that is ultimately sent to the user's destination.

Channels

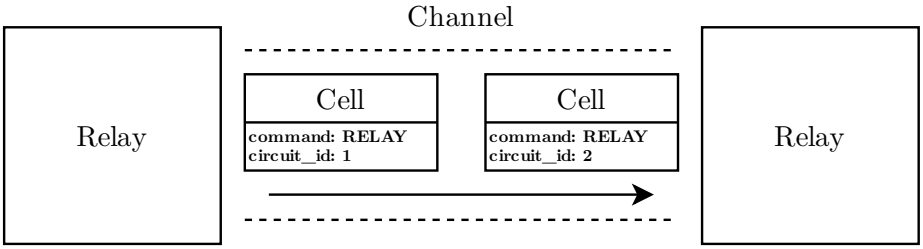

Figure 12: Cells are sent between relays over a channel. The circuit id allows to relay to know the next destination of the cell.

For two Tor nodes to communicate with each other, they set up a channel. A channel is responsible for transmitting cells from one node to another. Where a circuit is a path across 3 relays over which one client's cells are sent, a channel is the transport for all cells that are sent between two relays. As such, cells on a circuit will cross multiple channels, and a single channel will transport cells from many circuits. TLS channel is currently the only implementation of a channel within Tor's source code, however a QUIC-based implementation is created in this thesis.

To establish a TLS channel, a node creates a TCP connection to the next node in the circuit. Then, it establishes a TLS connection on top of the TCP connection. After this, the channel initiates a Tor link handshake. In this handshake, VERSIONS, NETINFO, CERTS and AUTHENTICATE cells are exchanged for version negotiation, validation of timestamps and IP addresses, and for client authentication. After the channel establishment completes, relay cells can be sent over the channel.

Each channel corresponds to a single connection. The channel is responsible for serializing cells unto the connection's output buffer and extracting cells from the connection's input buffer. The connection, on the other hand, is responsible for transmitting the data from its buffer to the destination node. Currently, this boils down to encrypting the data through TLS and handing it off to the kernel's TCP implementation.

Performance model

In this thesis, the performance Tor over QUIC is explored. There are a couple of factors to take into consideration when analyzing Tor's performance.

In an ideal scenario, Tor routes users' traffic with maximum bandwidth and minimum latency. Additionally, ideally, heavy downloaders aren't able to suffocate the bandwidth of small downloaders, such that some measure of fairness between users is maintained.

Tor is a network that forwards many TCP streams, divided into cells, through multiple relays. The performance of the network largely depends on the available

bandwidth of the relays. Additionally, latency, jitter and packet loss can impact routing performance. Given the relays, and the network they have available, the efficiency of the protocol leaves room for optimisation.

Between two relays, Tor circuits are multiplexed over a single TCP connection. Tor is designed like this such that relays can handle many connections at once. This allows for better performance and scalability than when using a single TCP connection per circuit. If a relay handles 1000 connections and 1000 circuits per connection, each with their own TCP connection, then one million open sockets would need to be handled. This is well above the limit for most computers and routers[26].

Even though multiplexing has a performance benefit over running many TCP connections simultaneously, multiplexing streams over TCP has its downsides. Head-of-line blocking will block the progress of all circuits between two relays when a single packet is dropped. This has a performance impact on all circuits that pass through these two relays. This effect becomes greater as the network is more lossy.

Tor uses a sliding window mechanism for end-to-end flow control. This makes sure that no more than a maximum number of cells are in transit at any given time. For every 100 cells that are received, the receiving end sends a `SENDME` cell back to the sender. This allows the sender to send another 100 cells. If the sender's window is exhausted, it has to wait with sending any more cells until more SENDME cells are received.

Currently, when a relay receives a cell, it is required to forward this cell reliably. If a relay drops a cell, the onion encryption fails, and the circuit is invalidated and broken down. Due to this restriction, a relay's buffers can fill up if it receives more cells than it can send. This leads to congestion at a relay, which increases latency throughout the network.

In 2017 Tor's default scheduler, which is responsible for the amount of cells that channels are allowed to send, was replaced by the KIST scheduler[27]. KIST throttles channels based on socket metadata it obtains from the kernel. This limits over-sending which improves circuit congestion.

3. Related work

Various research has been done to improve the performance of Tor. Performance issues roughly boil down to either inefficient multiplexing or inadequate end-to-end flow control.

The most relevant contributions are those that propose alternative transport protocols for Tor. Some background and categorization of these transport protocols is provided, then individual contributions are classified and discussed based on this categorization.

Multiplexing

When multiplexing multiple streams over a single TCP connection, packet loss can lead to head-of-line blocking. Tor is particularly effected by this as any two relays multiplex all shared circuits over a single TCP connection. This is currently a performance limitation within the Tor network.

Multiple solutions have been proposed to combat this in the form of alternative transport layer protocols. Caution needs to be taken with these proposals as ossification and lack of deployability become concerns for realistic usage of these protocols. Additionally, alternative transport layer protocols might expose new vulnerabilities that threaten anonymity.

End-to-end Congestion control

Tor's `SENDME` cell based end-to-end flow control has been proven to limit performance of the Tor network. Due to its long feedback time and simple window algorithm, and because cells are not allowed to be dropped once they're sent, relays can end up with long cell queues. This impacts both latency and throughput.

Additionally, overburdened relays impact the inter-circuit fairness within the network. Bulk downloaders send more cells, and get more bandwidth than web users within Tor. Adversely, web users benefit from lower latencies where bulk downloader don't. Various studies have proposed solutions for relay congestion within the Tor network to improve latency and fairness.

Hop-by-hop vs End-to-end

When it comes to proposals of alternative transport protocols, a major design decision is whether to use the protocol hop-by-hop or end-to-end.

Hop-by-hop

Hop-by-hop protocol connections are established between each two consecutive nodes in a circuit. This is a more standard approach since Tor uses hop-by-hop TCP connections in its network. Alternative hop-by-hop transport protocols aim to tackle the head-of-line blocking problem, as this is an issue that arises between nodes. However, end-to-end congestion often remains a performance problem.

End-to-end

Within Tor, end-to-end congestion control is managed by a simple sliding window algorithm, which has been shown to be inadequate and limit Tor's performance.

In end-to-end transport protocols, a connection is created between the initiator of the Tor circuit and the chosen exit node, instead of between each node. End-to-end transport protocols aim to improve on circuit wide congestion control. Alternative end-to-end transport protocols do come with additional costs. A circuit can only use this protocol if all nodes within the circuit support it. This makes deployment more difficult than replacement of hop-by-hop protocols, for which only two consecutive nodes have to support the protocol for it to be used. Additionally, with these protocols, there is a risk for metadata of the initiator to be leaked to the destination, this must be considered very carefully as the origin must remain anonymous for Tor to be effective.

Table 2: Alternative transport protocols for Tor

Paper	Protocol	HoL[1]	Fair[2]	Fp[3]	Depl[4]	Repr[5]
Vanilla Tor	hop-by-hop TCP + TLS	-	-	+	++	+
UDP Tor[6]	end-to-end TCP over UDP	+	+	-	-	-
DTLS[7]	hop-by-hop userspace uTCP over DTLS	+	+	+	-	-
uTor[8]	hop-by-hop kernelspace uTCP	+	+	+	--	-
QUUX[9]	hop-by-hop QUIC	+	?[6]	+	-	+
QUIC Tor[10]	hop-by-hop QUIC	+	+	+	-	-

[1]Solves Head-of-line blocking
[2]Whether a more fair distribution between bulk and light traffic is ensured
[3]Fingerprinting resilience
[4]How difficult it would be to deploy this protocol
[5]Reproducibility: whether the source code is available, the experiment can be re-run, and the results from the paper can be reproduced
[6]Fairness is not evaluated within the QUUX paper

UDP Tor (Viecco)

Viecco[6] proposed an alternative architecture to improve fairness between users in onion routing systems.

Rather than relays talking TCP to each other, the client establishes a TCP connection with the exit node. This connection is relayed over UDP, such that relay nodes can drop and reorder packets, the end-to-end TCP connection will handle congestion control. The exit node reassembles the TCP request and sends it to the server. Cryptography is adapted to support out-of-order delivery. Since the client's TCP is forwarded through multiple nodes, fingerprinting attacks remain. The study includes an experiment, but no link to the relevant source code could be found.

DTLS (Reardon)

Reardon et al[7] try to tackle the head-of-line blocking problem. Originally, Tor was designed to multiplex all circuits between two relays over a single TCP connection, to prevent running out of sockets on the host's TCP stack. This multiplexing caused congestion control and the head-of-line-blocking problems. Reardon's architecture uses a user space TCP stack instead, which is capable of handling many connections simultaneously. This allows Tor to dedicate a separate TCP connection per circuit. These connections are then made between hops on top of DTLS, a secure datagram based protocol. The study includes an experiment, but the source code is not included, as such the experiment cannot be reproduced easily.

uTor

uTor's[8] approach is similar to Reardon's. Unordered TCP is used to tackle the head-of-line blocking problem. uTor, however, does this through a kernel patch that allows `read()` calls to return a future region of the TCP stream out-of-order. The on-the-wire format could be unchanged, and only a small change to the Tor code has to be made, making this one of the more elegant solutions to the head-of-line blocking problem. However, the potential for deployment is severely limited due to the requirement of installing a kernel patch. Even though the paper features some experiments with their QUIC implementation within Tor, no available source code could be found. As such, the results of these experiments could not be verified.

DefenestraTor

DefenestraTor[28] aims to improve Tor performance by replacing its congestion and flow control by the N23 algorithm used in ATM networks. This is a credit based flow control algorithm that signals congestion through backpressure. The study shows that this algorithm helps combat congestion and improves response times and bandwidth of web usage within Tor.

QUUX

QUUX[9] is a Master's thesis featuring a QUIC implementation for Tor based on libquic, which is the QUIC library extracted from Chromium's source code[29]. Experiments with varying loss are done through both Chutney and Shadow. A significant performance advantage over vanilla Tor is shown at various loss levels. Source code and experiment sources are included with the paper. An attempt at reproducing the results of the paper have been done, but unfortunately the modified Tor code resulted in compilation failures that were difficult to remedy.

The extracted libquic library has not been in active development, and the IETF QUIC specification has evolved since then. As such, it is valuable to implement Tor over QUIC with a more recent, actively developed, QUIC library.

Adding QUIC support to the Tor network

In this master's thesis[10], TCP and TLS are replaced by QUIC through Cloudflare's Quiche library. QUIC is used on a hop-by-hop basis. Support for different streams and built-in TLS ensure for authenticity and prevents the head-of-line blocking problem. The study shows some promising Time to First Byte results in its experiments. Unfortunately, the prototype implementation would sometimes stall, leading to worse latencies and bandwidth than vanilla Tor. Additionally, limitations in the used network simulators, Chutney and Shadow, prevent large scale network simulations to be run.

Mind the Gap

Besides work on transport protocols, research on Tor performance has been conducted on other areas as well.

Tschorsch and Scheuermann's "Mind the gap" paper[5] proposes BackTap, a backpressure-based flow control algorithm that replaces Tor's end-to-end sliding window algorithm. It's shown that this algorithm has a great impact on relieving congestion on the Tor network. When using backpressure in consecutive relays, rather than the current end-to-end flow control algorithm, the feedback loop for congestion is shorter, and congestion can be dealt with more swiftly. Additionally, the study claims to improve fairness, both between Tor circuits and between Tor traffic and unrelated network traffic.

KIST

Jansen et al. propose KIST[27], which attributes congestion in the Tor network to the fact that relays write too much data to their sockets' output buffers, which causes increased queueing delays. Additionally, sockets are written to sequentially, rather than following the relay's circuit priority. This means that the prioritization of circuits is effectively relinquished from Tor code to the kernel's TCP implementation. KIST is an alternative scheduler for Tor which uses socket metadata to make informed decisions on how much data should be sent to its sockets. This approach has shown to reduce congestion and improve latency

throughout the Tor network. The KIST scheduler has since been implemented in Tor.

4. Design and Setup

Between two Tor relays, all circuits are multiplexed over a single TCP connection. This comes at a performance cost. Due to head-of-line blocking, a single dropped packet will block all streams between two relays. Additionally, TCP's congestion control will throttle all streams when a congestion is detected, this favors bulk downloaders over users with light usage and affects fairness. Integrating QUIC into Tor has the potential to remedy these issues due to QUIC's built-in UDP-based multiplexing and its per-stream congestion control.

When replacing Tor's inter-relay TCP connections with QUIC, certain design decisions have to be made. Namely, QUIC can be implemented as transport between relays, or as end-to-end protocol between the client and exit node.

4.1 Design

Goals

To improve on both vanilla Tor and previous work, certain design goals are established. Most of these goals are similar to those in previous work on Tor over QUIC. However, reproducibility is added as a goal. This will help future research to build on top of this thesis.

Head-of-line blocking

As explained in chapter 2, Tor suffers from head-of-line blocking. This limits the efficiency in which TCP streams can be multiplexed between Tor relays.

Within the QUIC design, head-of-line blocking only happens within individual streams, not on the connection level. Tor circuits can be mapped to QUIC streams based on their circuit id, such that head-of-line blocking of a single circuit will not impact other circuits on the same connection. QUIC should be implemented in this way, such that head-of-line blocking, as it currently occurs within Tor, is solved.

Fairness

Due to Tor's multiplexing over TCP, TCP's congestion control applies to all circuits between two Tor relays as well. This impacts fairness between circuits, as TCP's backoff will lower the transmission rate of the entire connection. For example, if a bulk downloader and a light user of Tor use circuits that go through

the same congested link, both their circuits will be throttled proportionally. This leaves the light user with less bandwidth because they put less load on the network, while ideally, bandwidth is distributed equally amongst users. By relying on QUIC's per-stream congestion and flow control, fairness should be improved in the QUIC implementation. Jain's fairness index[30] will be used to calculate the fairness of the implementation.

Privacy and security

Providing privacy and security is essential for Tor to operate. Users use Tor such that they can increase their privacy on the Internet. If Tor's security were to be compromised, then this would affect its users' privacy as well.

Correlation attacks are important to consider when designing an alternative transport protocol for Tor. An attacker could observe both ends of a circuit, and de-anonymize users based on patterns of bandwidth, congestion and protocol metadata.

Numerous attacks on Tor have been shown, of which some based on fingerprinting of TCP's metadata. Especially if QUIC is used end-to-end, fingerprinting must be mitigated specifically.

The Tor over QUIC design should not allow attacks on privacy to happen that are not possible within Tor currently. To prevent metadata leakage between the client and the exit node, QUIC is integrated in a hop-by-hop fashion instead of end-to-end. Because most of QUIC's headers are encrypted, QUIC exposes less information than TCP. For example, on top of the headers that UDP have in common, TCP exposes a sequence number, acknowledgement number, flags, window size and options. UDP does not expose additional information over TCP, and QUIC only exposes a connection identifier and a limited amount of flags. QUIC's encryption of other headers should improve against attacks that use transport protocol headers for attacks against Tor.

Deployability

The Tor over QUIC design must be deployable. Ossification puts a limitation on which transport protocols can be deployed successfully. Since middleboxes tend to block less popular transport layer protocols such as SCTP, it is desirable to use either TCP or UDP. Additionally, ossification must also be considered when using TCP extensions, as those can cause middleboxes to deny packets as well.

Previous research, such as uTor[8], proposes a transport protocol that requires a kernel patch to run. Kernel patches are difficult to install and likely need manual intervention to install. It would take a long time before a decent share of Tor relays can patch their kernel to allow for an alternative transport protocol to be deployed. It is preferred to use a protocol based on TCP or UDP, as these are already widely available in kernels across modern operating systems. Alternatively, a userspace protocol could be used, such that the protocol can be upgraded along with Tor itself.

If QUIC is implemented end-to-end, then every relay within a circuit needs to be updated before the new protocol can be used. This is a drawback that makes QUIC over Tor difficult to deploy as well. Since relays update their Tor version

gradually, it is preferred to optimistically support a new protocol on a per-link basis. Such that only two relays need support for the new protocol to connect to each other through it. This would greatly increase the pace at which the new protocol can be deployed at scale.

As such, QUIC is integrated in a hop-by-hop fashion. The kernel's UDP implementation is used. On top of this, the QUIC protocol can be included into Tor in the form of a library. This allows QUIC to be updated easily along with Tor updates.

Reproducibility

Only few of the previous studies contain access to the source code that is used to run their experiment. This makes it difficult to reproduce the experiment and verify the results. Additionally, the implementation has to be redone when building on top of previous research. Only few of the proposed alternative transport protocols contain enough resources to reproduce the published experiments.

The predecessor of this paper[10] contains an experiment based on an implementation of QUIC over Tor. Unfortunately, the source code and test setup for this experiment could not be obtained. Because of this, the implementation effort that made the thesis possible must be redone to build on top of its work.

To streamline this process this in the future, an additional requirement will be reproducibility. Through the use of publicly hosted source code, and inclusion of any scripts that are required to run the experiment, the experiment can be made fully reproducible.

The source code for the Tor fork, which integrates with QUIC, can be found at GitHub[31].

Adding QUIC to Tor

QUIC is generally a good fit for an alternative transport protocol for Tor. QUIC's streams can map unto Tor's circuits which will remedy the head-of-line blocking problem. QUIC's per-stream congestion control provides the means to improve fairness.

Because QUIC is used in HTTP/3, it is already deployed widely globally. Ruth et al.[32] showed in 2018 that 2.6% to 9.1% of all Internet traffic uses the QUIC protocol. This inspires confidence for the deployability of QUIC within Tor.

End-to-end or Hop-by-hop

Two major designs exist for using Tor over QUIC: end-to-end and hop-by-hop.

In the end-to-end design, a QUIC connection is made between the two edges of the Tor network, spanning across relays. The individual QUIC packets are then forwarded between hops over UDP.

Tor currently offers no end-to-end congestion control. Because of this, many cells can get stuck at a single congested relay, which degrades the overall bandwidth of all circuits that pass this relay. Using QUIC end-to-end such that its flexible congestion control will limit the amount of cells sent by the client would solve

this issue. However, the end-to-end design comes with security and deployability issues that make it infeasible to use.

In end-to-end designs, the headers and parameters of the protocol will be sent from the client to the Tor exit node. An adversary that observes both ends of this connection could relate this information and de-anonymize a user. Relating metadata, re-ordering packets and explicit congestion control information can all be used to create a covert channel.

For example, if an adversary controls the first relay in a circuit, and an end-to-end QUIC design is used, the adversary can send encoded messages to the exit node. This can be done by reordering the packets in a specific way. Because QUIC is UDP based, and used end-to-end, the relays on the path will forward the UDP packets without reordering them. Through this, the modified ordering from the first relay on the path can persist throughout the circuit until the exit node is reached. This reordering could be used to encode the address of the user[33]. Vanilla Tor doesn't have this problem as TCP is used between relays. Because of TCP's in-order delivery, each relay will ensure all packets are sent in the order the client initially sent them.

The end-to-end information leakage is a problem for Tor's anonymity requirement that is not easy to circumvent.

In end-to-end designs, QUIC packets are forwarded through the Tor network over UDP, encrypted with DTLS. Because DTLS is not a reliable transport, packets that are dropped between relays must be retransmitted from the client, even if the packet drop occurs just before the exit node. Since relays of a Tor circuit are chosen to be in different jurisdictions, there is a much higher latency than in typical non-Tor QUIC connections. This extra delay for retransmitting will negatively affect the performance of the end-to-end design.

In hop-by-hop designs, a QUIC connection is made between each pair of relays. This means that both TCP and TLS as they are integrated in Tor now are both replaced by a QUIC connection with TLS built-in. In hop-by-hop designs, Tor circuits are mapped to QUIC streams, this allows for efficient multiplexing without head-of-line blocking. Additionally, QUIC allows for flow control and congestion control to happen both on a per-stream and a per-connection basis. This can solve the lack of fairness between circuits within Tor, dedicating an equal amount of bandwidth to each QUIC stream.

Hop-by-hop designs are advantageous for deploying as well. Rather than each relay within a circuit, only two connected relays need to be updated to a Tor version that supports QUIC to take advantage of its benefits. Since it can take some time for relays to update their Tor version, this more flexible requirement will make deployment beneficial sooner.

This design can save on complexity as well. Within the end-to-end design, the TCP and TLS stack need to be replaced by UDP and DTLS to allow for out-of-order forwarding of UDP packets while retaining encryption between relays. On top of that, a QUIC connection needs to be made between the client and the exit node. In a hop-by-hop design, only the TCP and TLS stack need to be replaced by QUIC.

0-RTT

TLS 1.3 introduces a feature called "zero round trip time connection resumption" (0-RTT), which allows the client to send user data immediately, effectively lowering the handshake latency to zero. This can be done when the client has saved a Session Ticket that it has obtained through an earlier connection with the server. The data can be encrypted and sent along with the TLS handshake data immediately, without having to wait for a response from the server.

Since QUIC integrates with TLS 1.3, this feature can be enabled in QUIC. Additionally, QUIC adds to this latency reduction by providing zero round trip time connection establishment. Where both TCP and TLS would both add latency through their own handshakes, QUIC can bring this down to zero.

However, by enabling 0-RTT in TLS 1.3, replay attacks become a possibility. When an adversary eavesdrops a connection and records the data of a 0-RTT handshake, it is able to replay this data to the server. The server will not be able to distinguish between the original sender's packet and the adversary's

Because Tor aims to provide a high level of anonymity and security, it is unwanted to become vulnerable to replay attacks. Additionally, the connections between relays are long-lived, and fast connection establishment has only a limited impact on performance. 0-RTT will be turned off for Tor, and a single round trip will be required for the connection handshake.

Fallback

Because Tor relays are operated by volunteers, it is not feasible to update the Tor version of all relays at once. To prevent disruptions in the network when deploying Tor over QUIC, a fallback needs to be available. When two relays both support QUIC, a QUIC connection is established. When at least one of the relays does not support QUIC, the relays must connect over TCP + TLS. This distinction can happen primarily through advertising. Tor's directory servers keep track of relay metadata. QUIC support could be a new field within these directories, such that the availability of QUIC becomes well-known. This will allow for gradual deployment of QUIC over Tor.

QUIC libraries

There are a number of open-source QUIC implementations that could be used for integration with Tor.

Table 3: Popular QUIC libraries

Name	License	Language	Owner
Chromium[34]	Free	C++	Google
mvfst[35]	MIT	C++	Facebook
ngtcp2[36]	MIT	C	ngtcp2
Quinn[37]	Apache	Rust	Quinn-rs
Neqo[38]	Apache	Rust	Mozilla
Quiche[39]	BSD 2	Rust	Cloudflare

Chromium

Before QUIC development was transferred to the IETF, Google had initially designed and developed the QUIC protocol. The effectiveness of this protocol was tried out in the Chromium browser. The experiment was successful and Chromium QUIC protocol now conforms to the IETF specification. QUIC in Chromium is production-ready and deployed globally. However, this implementation contains little publicly available documentation which makes it difficult to use[34]. Since the implementation shows little dedication to support third party usage, breaking changes might occur without notice.

Mvfst

Mvfst is a C++ QUIC implementation created by Facebook. It aims to be performant and has been tested on a large scale[35]. The implementation's API is still in alpha phase. This makes it unfit for integration with Tor.

ngtcp2

ngtcp2[36] is an open source implementation of the QUIC protocol, written in C. It has support for multiple cryptography library backends. Its documentation is limited, and the project is not backed by a company which might impact the project's longevity. No claims of production-readiness are made.

Quinn

Quinn[37] is an open source QUIC implementation written in Rust. It differentiates itself through its Futures-based asynchronous API. Futures are a paradigm of the Rust language that can simplify working with asynchronous operations. However, since Tor is written in C, and the library's core focus is asynchronous programming in Rust, this library is not a great fit for Tor integration. The library has a decent amount of documentation, but there is no mention of interoperability with C.

Neqo

Neqo[38] is Mozilla's QUIC implementation, used in the Firefox browser. It is written in Rust, and it's deployed widely within Firefox. Like Chromium's implementation, Neqo contains little documentation, and is mainly focused on integration with its browser.

Quiche

Quiche[39] is a QUIC implementation in Rust backed by Cloudflare. This implementation is flexible in its usage because it leaves I/O operations such as socket and event loop usage to the application. Quiche contains more documentation than alternative implementations, and integrates with well known third party applications such as Curl and NGINX. Additionally, Quiche has good support for C interoperability, which makes it attractive for Tor integration.

Cloudflare is a company that provides web infrastructure such as content delivery, DNS and DDoS protection. Quiche is used in production for Cloudflare's edge

network.

Quiche is distributed under BSD 2-clause license which is compatible with Tor's BSD 3-clause license.

Due to its flexibility, reliability, good support for third party usage and license-compatibility, Quiche is a good fit for integration with Tor. As such, this is the library that is used to implement Tor over QUIC.

4.2 Implementation

Integrating Quiche with Tor is not a straight forward process. Ideally, separation of concerns is used, and only the part of the code that handles the transport layer protocol has to be changed. This is unfortunately not the case. Implementation difficulties, and the decisions that are made to overcome these, are laid out in this section. The source code of the fork can be found on GitHub[31].

Background

Some familiarity with the general structure of Tor's codebase is required to understand the issues that come with QUIC integration. The most relevant abstractions are channels and connections.

Abstractions

Tor's source code contains abstractions that use an object-oriented style to define classes and subclasses. Macros are used to convert a class to its subclass and vice-versa.

For example, the `connection` class has a subclass called `or_connection`. Where `conn` is a reference to a `connection` , `TO_OR_CONN(conn)` can be used to convert a `connection` to its `or_connection` counterpart.

This subclass mechanism is used for `channel`s as well as `connection`s.

Connections

The `connection` class deals with reading and writing bytes over connections. There are several subclasses for `connection`, such as `listener_connection` and `or_connection`. Each `connection` aims to represent either a TLS connection, a TCP socket, a unix socket or a UDP socket.

The OR connection subclass (`or_connection`) reads and writes data over TLS to another relay. This subclass handles all onion routing cells.

Channels

Channels are a higher level abstraction than connections. Where connections use buffers to read and write data, each `channel` owns a single `connection` which it uses to transmit cells to another relay. Currently, the only subclass of channel is TLS channel (`channel_tls`), which uses an `or_connection` internally.

The OR connection is responsible for encoding cells and sending them over a TCP connection. The TLS channel sits between its superclass, `channel`, and its OR connection. The TLS channel communicates with `channel` by implementing the functions in `channel`'s interface. Beyond passing cells from the upper layer to its `or_connection`, TLS channel adds a layer of authentication on top of TLS through the Tor link handshake. This handshake uses `AUTHENTICATE` and `CERTS` cells to verify the Tor identity of the relay that it is connected to.

QUIC channel

To integrate QUIC with Tor, the Quiche[39] library is used. Quiche is written in Rust, but it provides C bindings through a single header file. Where Tor's TLS channel uses OpenSSL[40] to implement TLS, Quiche uses BoringSSL[41] for its TLS implementation.

Because QUIC can only be used when two subsequent relays on a path support it, QUIC and TLS must be able to run alongside each other. When QUIC is not supported by two relays, the TLS channel can then be used as fallback.

There are two ways in which Quiche can be added to Tor: by creating an alternative channel class, and by creating an alternative connection class.

Intuitively, as QUIC functions as a transport layer protocol, it makes sense to use TLS channel, and split its OR connection into two implementations: the original TLS over TCP connection and a new QUIC connection. However, the design of TLS channel is incompatible with Quiche in two ways.

First, TLS channel uses a listener connection. Internally, this is a server-side TCP socket that is bound to the onion routing port (`ORPort`). When a client connects to this socket, a new TCP socket is created, this socket is used to create the corresponding OR connection. This model is incompatible with QUIC because UDP sockets are connectionless. Within QUIC, an incoming UDP datagram needs to be parsed by QUIC to know the corresponding client. In TCP, the client is known based on the socket on which the packet is received. Where TLS channel exposes a function that takes an `or_connection` from a known client, for QUIC support a separate function that receives a datagram from unknown origin is required.

Additionally, TLS channel accesses OR connection's internal TLS certificate in its Tor link handshake. Quiche uses TLS internally, but doesn't expose the certificates that are exchanged. This warrants a change to the Tor link handshake such that it can operate irrespective of the TLS handshake certificates.

Because of these fundamental differences, it is decided to create an alternative channel, rather than an alternative connection. Through this, QUIC channel can operate parallel to TLS channel. The channel superclass provides a number of functions that are implemented by both QUIC channel and TLS channel, and the channel superclass decides which of the subclasses is used.

It is possible to create a connection subclass for QUIC connections, similar to how TLS channel uses an OR connection. However, the connection class contains a single input buffer, and a single output buffer. This does not translate well to QUIC which requires a buffer per stream. Due to this lack of multiplexing

support of connections, and because Quiche already implements most of the responsibilities of the connection class, it is decided to not create a QUIC connection subclass. Instead, QUIC channel uses Quiche directly. This avoids complexity caused by interface incompatibility with the connection class. An overview of the resulting architecture of QUIC channel can be seen in figure 13.

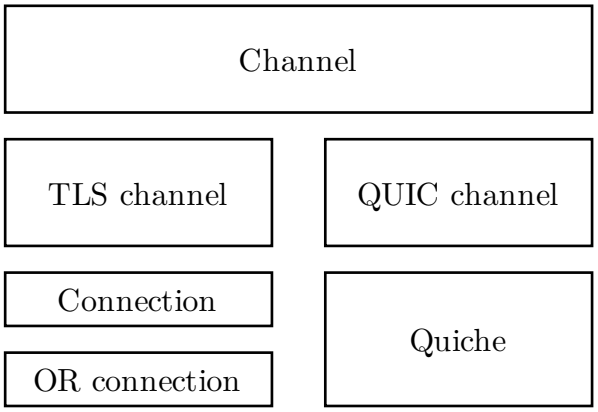

Figure 13: Channel Implementation

Routing

Tor uses an event loop to keep track of when sockets can read and write efficiently. When data arrives at a socket, an event is fired. In vanilla Tor, the TCP socket that data arrives at corresponds to an OR connection. This OR connection is then passed to a function of its corresponding TLS channel.

Because UDP is a connectionless protocol, and a UDP socket is typically not bound to a specific source, an alternative approach is used for QUIC. Since channels correspond to a QUIC connection, rather than a socket, an incoming datagram must be parsed by QUIC before its corresponding channel can be found. As such, QUIC channel provides a function that is called whenever incoming UDP data is received on the relay's ORPort. The incoming datagram is parsed, and if the connection id of the QUIC packet is known, the corresponding channel is found in a hashmap. Otherwise, a new QUIC channel is instantiated, and inserted into the map.

QUIC streams

To prevent head-of-line blocking between Tor circuits, each circuit is assigned to a stream. A hashmap is used to map circuit ids to QUIC stream ids. If a cell with an unknown circuit id is sent or received by a channel, a new QUIC stream id is assigned to this circuit. Since the QUIC specification requires stream ids to be monotonically increasing, there is no choice to be made for which stream id to pick.

There could be concerns regarding the amount of extra information that is exposed by using a stream id per circuit. However, QUIC's stream ids are encrypted, QUIC is used in a hop-by-hop fashion, and relays already know the

circuit ids of the cells they forward. Additionally, relays know how many circuits are forwarded between each other. As such, using monotonically increasing QUIC stream ids between relays does not expose extra unknown information.

Buffers

Each OR connection contains a single input and output buffer. However, QUIC's multiplexing necessitates a buffer per-stream, rather than per-connection. This allows streams to move independently, solving head-of-line blocking for Tor circuits. A hashmap is used to map a channel and a stream id to their corresponding buffer.

KIST Scheduler

In 2017, introduced the KIST scheduler to replace its vanilla scheduler. KIST schedules cells on connections based on kernel information of sockets. Unfortunately, this method relies on TCP, and cannot be used for UDP without major adaptations. Because of this, when using QUIC, KIST's fallback scheduler, KISTLite, is used.

KISTLite is still an upgrade over the vanilla scheduler, but leaves for some performance improvement. In future work, the KIST scheduler can be adapted to work with QUIC as well.

Scheduler interface

QUIC features per-stream congestion and flow control, and QUIC streams are mapped to Tor circuits. Tor's scheduler uses channel's interface to find out how many bytes are queued and how many cells can be written. After deciding how many cells the channel should write, it pops the cells from a chosen queue.

The amount of cells that QUIC channel can write, depends on which circuit the cells belong to i.e., which QUIC stream is used. To take this into account, channel's interface needs to be changed, and the way the scheduler works will then depend on which channel type is used.

This change is out of scope of this thesis. Instead, a more simple approach is used. The sum of the sizes of the output buffers of the QUIC connection is used. Only as many cells as fit within a constant high-water mark are allowed. When enough cells are flushed, and the sum of queued output bytes is lower than a constant low-water mark, then more cells are requested. This approach is similar to how TLS channel operates. However, to obtain optimal performance while operating multiple streams, the low-water and high-water marks are increased compared to TLS channel. While this change made a significant impact for QUIC in a rudimentary performance benchmark, increasing the low-water high-water marks in vanilla Tor did not lead to a performance improvement. As such, QUIC requires a larger buffer size to reach optimal performance.

Identity cells

Within the Tor link handshake, the TLS certificates are used to obtain and verify an identifier for the relay on the other side. Currently, it's not possible

to access these certificates when using Quiche. A workaround is used to make the Tor network operate, however this approach is not secure for production use. In a future version, the Quiche library must allow access to used certificates. Alternatively the Tor link handshake can be made independent of the TLS encryption that its connection uses.

OpenSSL and BoringSSL

Quiche is written in Rust, and C bindings are provided as a single header file. To use Quiche in a C project such as Tor, normally a static library is created, and included in compilation. However, Tor uses OpenSSL and Quiche uses BoringSSL. BoringSSL is a fork of OpenSSL with no compatibility guarantees. As a result, including Quiche in Tor as a static library causes conflicts. To remedy this, Quiche is built as a shared library instead.

4.3 Simulation

To test the effectiveness of an alternative transport protocol within Tor, a Tor network has to be simulated. To do this, most commonly, Chutney or Shadow are used.

Shadow

Shadow[42] is a network simulator that is focused on simulating Tor on a single machine. Even though Shadow supports simulating the network for other applications, its main focus is Tor.

Instead of using the operating system, Shadow runs application code directly while simulating system calls. This allows it to simulate networks at a larger scale. Shadow can be used to run thousands of network connected processes on a single machine.

Unfortunately, Shadow has limited support for UDP, this makes it difficult to use QUIC in this simulator.

Chutney

Chutney[43] is a network simulator that runs multiple Tor instances on the host operating system. It consists of a set of scripts that set up a test network and allows for some basic monitoring.

Tor instances that are launched by Chutney address each other over the loopback interface. As such, a unique port number is assigned for each function of each instance.

Chutney is useful for testing a Tor network efficiently, but it lacks support for adding latency and loss between instances. A separate bridge can be used to add latency, but packet loss is still not supported due to the way TCP retransmit works within Mininet[10].

Mininet

Mininet[44] is a tool that creates a virtual network that runs real kernel, switch and application code. It can run on a single machine, and allows for creating various network topologies. Through Mininet, links between nodes can be customized. Bandwidth, loss, latency and jitter can be set on each individual link within the virtual network topology.

Within Mininet, custom network topologies can be created in the form of a Python class. Programs on the host operating system can be executed from a node within Mininet. In this case, the process will use the virtual network to interact with the other nodes.

Containernet

Containernet is a fork of Mininet that uses Docker containers rather than the operating system as its hosts. Using Docker has the advantage of running a reproducible operating system environment. Instead of reinstalling binaries on the host operating system to change a program, Docker containers can be swapped out easily without affecting the host.

Containernet can be configured through Python. A network topology is defined with hosts as Docker containers and links between them. The Docker containers can be created with limited memory or CPU, and the links can have bandwidth, latency, loss and jitter restrictions imposed on them.

For Tor, this means that the efficiency of transport protocols can be evaluated, with different levels of memory, CPU, bandwidth and latency limitations. This is vital for testing the effectiveness of congestion control within the Tor network.

Additionally, the usage of Docker for the hosts, and Python for the network topology, the entire experiment can easily be made reproducible. Docker containers can be shared such that peers can run an exact replica of the hosts used in the experiment.

In this thesis, a Tor configuration that works with Containernet is generated, and a Containernet setup with restricted links between hosts will be used to discover performance characteristics of Tor running over QUIC.

Containernet's adaptions for simulating Tor, which is used for the experiments in this thesis, along with tooling, can be found at GitHub[45]. Because both the implementation of QUIC within Tor and the network simulation infrastructure are open source, reproducing the results that are found in this thesis should be possible. This aims to satisfy the reproducibility goal.

5. Performance evaluation

To evaluate the performance of Tor over QUIC, experiments are done both through chutney and Containernet. Specifically, download throughput, Time to First Byte (TTFB) and Time to Last Byte (TTLB) of different Tor setups are measured.

Environments with varying bandwidth, latency, loss and jitter are used to compare the impact of vanilla Tor and Tor over QUIC. Additionally, the way in which several circuits interact when their paths share a relay is explored.

A study from 2010[46] contains guidelines for how accurately model Tor performance evaluation. Later studies[27][9] use this as a baseline for their experimentation. In this thesis, this trend is continued. Namely, Tor usage is categorized into two types of workloads: web usage through HTTP and downloading through BitTorrent. Bittorrent is not recommended by Tor[47] as a user's IP address can still be leaked by BitTorrent clients. However, the study found that BitTorrent usage attributed to 5% of Tor clients, and 40% of the total amount of bandwidth in Tor.

A web usage workload is modeled as a small download of 320 KiB, and a bulk downloading workload, such as BitTorrent, is modeled by a larger request of 5 MiB. Even though websites have grown in size, and are often larger than 320 KiB now, this is still the used size such that results can be compared with previous studies on Tor.

For the experiments, an initial download is done such that a Tor circuit is created before the experiment is started. This ensures that no added latency due to circuit creation is found in the results of the experiments. Circuit creation time is not measured as this happens infrequently, and thus has a small impact on quality of experience.

In these experiments, by default, QUIC uses CUBIC as its congestion control algorithm and Hystart++ enabled. This conforms to Quiche's default settings, and recommended usage. Vanilla Tor uses CUBIC as well, as this is the default congestion control algorithm for TCP within Linux.

Measuring bandwidth

Bwtool, a bandwidth measurement tool, is created to perform these experiments. Bwtool serves a file of given size over HTTP, and then downloads this file over

a given SOCKS proxy i.e., a Tor client process. Bwtool measures the TTFB, TTLB, and the time of reception of each chunk of given size.

Bwtool integrates with SOCKS, rather than HTTP, because HTTP clients tend to not expose the time of reception of the response's first byte. Instead, HTTP clients receive and parse HTTP headers, and then expose the HTTP body as a byte stream[48][49]. Bwtool records the first byte of the entire HTTP response, rather than the first byte of the HTTP body.

Note that when an experiment on a file of, for example, 320 KiB is done, that the actual download size is 320 KiB plus the size of the HTTP headers. This means that roughly 204 extra bytes are downloaded in each of the experiments.

The use of SOCKS itself is not expected to have a big impact on the results. SOCKS is a lightweight protocol, and the Tor client typically runs on the same host as the application that connects to it. Within these experiments, Bwtool, which is the SOCKS client, and the Tor client, which is the SOCKS server, run on the same host as well. As a result, SOCKS imposes a minimal amount of restrictions to latency and bandwidth. A simple test is done to confirm this. A file of 1 GB is downloaded by Bwtool, through a local SOCKS proxy. The resulting TTFB is 4 ms and the average throughput is 357 MB / s. This provides confidence that SOCKS has a minimal impact on the results of the experiments in this thesis.

Containernet

For Containernet experiments, a star topology is used. There is a single switch, and each host is only connected to this switch. Each link imposes a set amount of latency. A minimal Tor network is used, which contains 3 directory authorities, 3 exit nodes and 3 clients. The bandwidth of each link is capped at 100 MiB per second, as is the case in the Tor modeling study[46]. This is chosen as a high amount of bandwidth such that it is not the bottleneck of download performance within Tor.

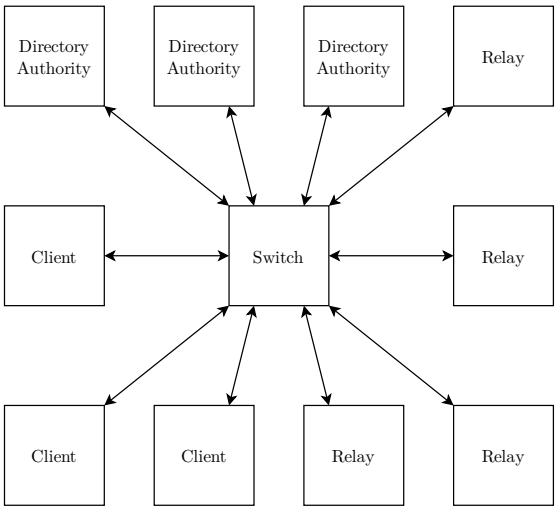

Figure 14: Tor Test Network setup using a star topology. Links are denoted by arrows. The links have limited bandwidth, and imposed latency. Two links are traversed between each pair of hosts.

In some scenarios, latency is applied to each link. Note that if a single link has 10 ms latency, each two hosts have two links between them, such that a packet takes twice the per-link latency to arrive at the next host i.e., 20 ms. A Tor circuit contains 3 hops, and Bwtool acts as both the client and the server. Bwtool resides in the same Docker container as the Tor process it uses. In total, 4 edges, or 8 links, are traversed to send a packet to the server through Tor. This means that a round trip over Tor bridges 16 links. Bwtool uses HTTP over TCP over SOCKS. To calculate the TTFB, two round trips are necessary: one for the TCP handshake and one for the HTTP request itself. This means that the minimum TTFB is equal to 32 times the latency of an individual link in the Containernet topology.

Since the round trip time of a Tor circuit averages around 400 ms, and 16 links are passed for a single circuit round trip, a link latency of 25 ms is chosen in scenario 4, 5 and 7.

Scenario 1 - Containernet Baseline

For the experiments in this thesis Containernet is used for its flexibility. However, Containernet has not been used before for performance analysis of Tor. To detect significant performance regressions caused by Containernet, Chutney is compared to Containernet.

To make this comparison, a basic experiment is run on both simulators. A small Tor network is run, and Bwtool is used to measure the bandwidth of a 320 KiB download over the network. No restrictions are imposed to the bandwidth, latency or loss of the links. The TTFB and TTLB are measured throughout 100 runs in each simulator.

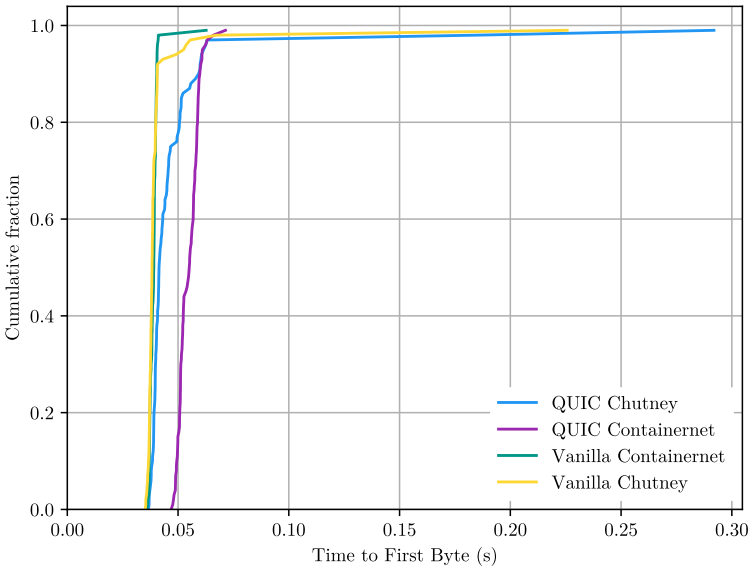

Figure 15: Baseline - TTFB 320 KiB, uncapped

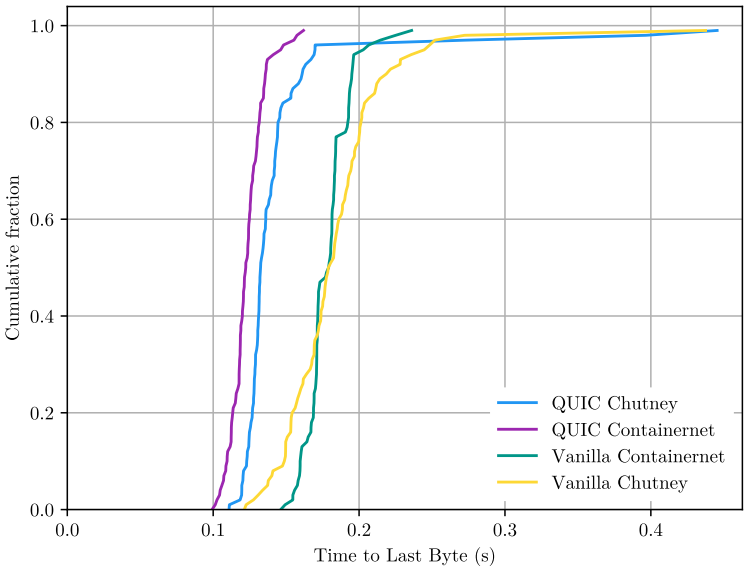

Figure 16: Baseline - TTLB 320 KiB, uncapped

Table 4: Baseline - Average (standard deviation) of TTLB

Baseline	.
QUIC Chutney	0.143s (σ 0.044)
QUIC Containernet	0.123s (σ 0.011)
Vanilla Chutney	0.183s (σ 0.038)
Vanilla Containernet	0.178s (σ 0.015)

In figure 15, it can be seen that the TTFB is similar in both experimental setups. The TTFB for QUIC within Containernet is slightly higher overall, and the TTFB for QUIC within Chutney is slightly less consistent than both vanilla measurements. On the other hand, the TTLB in figure 16 shows that QUIC has a lower TTLB than vanilla within both network simulators. Overall, the bandwidth in Containernet is slightly higher.

Since no latency or loss restrictions are applied in these benchmarks, performance is mainly limited by CPU and the host's networking performance. In further experiments, it is expected that restrictions on the network links do become bottlenecks for Tor throughput. Overall, the simulators provide similar results in an ideal scenario, in which simulator inefficiencies are exposed the most. There is confidence that Containernet is suitable for performing at least small-scale simulations, especially when links are restricted, such that these restrictions become the performance bottleneck.

Scenario 2 - Latency

Next, the impact of latency on both transport protocols within a minimal Tor setup is explored. A file of 320 KiB is downloaded through Bwtool within Containernet. For this experiment, a latency of 10 ms, 25 ms and 50 ms are applied to each link within the topology. This is based on the circuit round-trip latencies from Tor's website for performance metrics[50]. Note that there is no loss introduced to this environment. As such, congestion control algorithms might have minimal impact on the measured performance. To explore this, QUIC Reno, which replaces CUBIC with the Reno congestion control algorithm, is added as an alternative in the experiments.

The TTLB and TTFB are recorded throughout 100 runs for both vanilla and Tor over QUIC with CUBIC and Reno. While QUIC has Hystart++ enabled by default, the label "nh" indicates that Hystart++ is disabled.

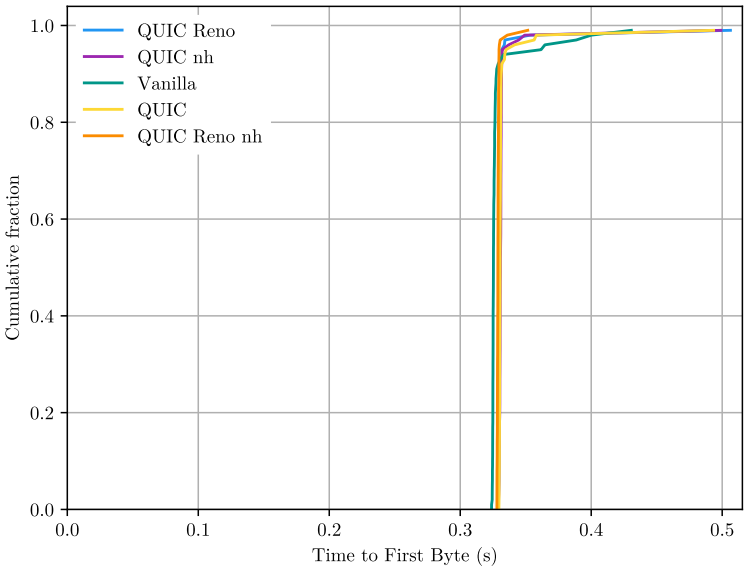

Figure 17: Latency - TTFB 320 KiB 10 ms latency

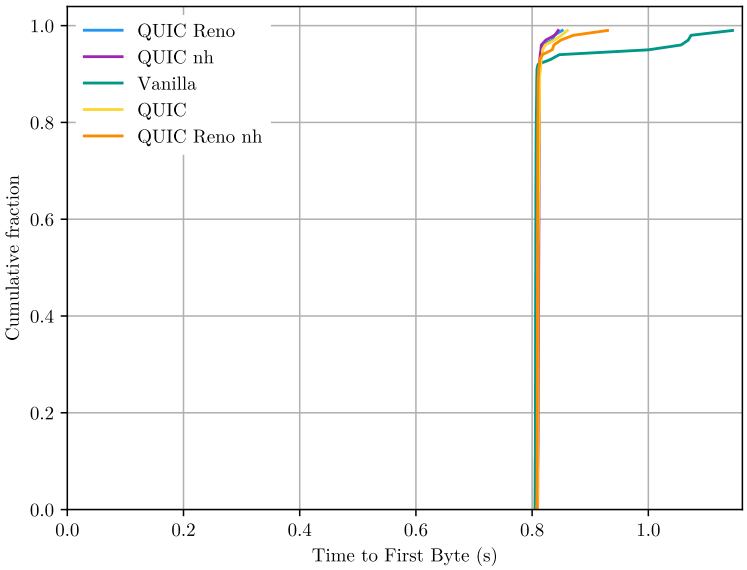

Figure 18: Latency - TTFB 320 KiB 25 ms latency

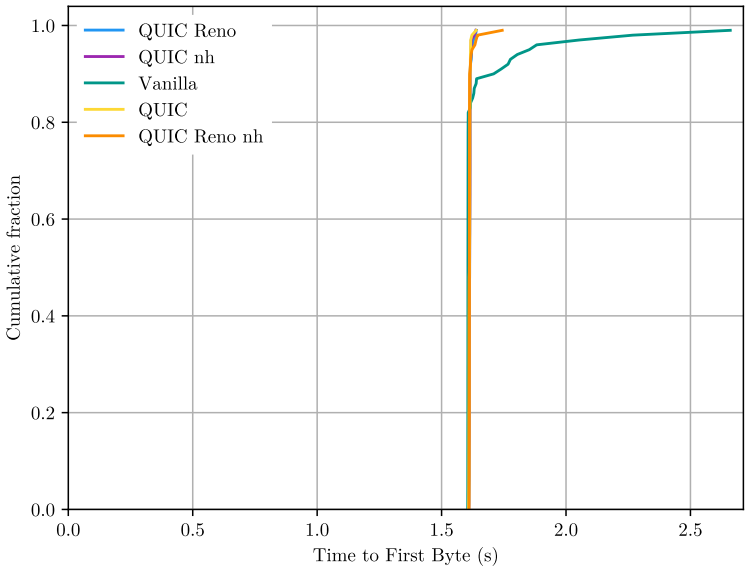

Figure 19: Latency - TTFB 320 KiB 50 ms latency

As explained earlier, the theoretical minimum of the TTFB is 32 times the latency that is applied to each link. These minimums for 10 ms, 25 ms and 50 ms are 320 ms, 800 ms and 1600 ms respectively. In figures 17, 18 and 19, the 90th percentile of the test runs are close to these theoretical minimums. This means that both vanilla Tor and Tor over QUIC open connections through the network efficiently. For the 10th percentile slowest TTFB results, QUIC performs slightly more consistently than vanilla, especially in the 50 ms latency experiment.

In figure 17, observe that vanilla Tor's TTFB is generally 0.01 second lower than Tor over QUIC' s TTFB. This corresponds to the results seen in figure 15. Oddly, this did not appear when running QUIC over Chutney. Due to the complexity of the simulators, it is difficult to pinpoint the cause of this difference. Due to its relatively small impact, it has not been investigated further.

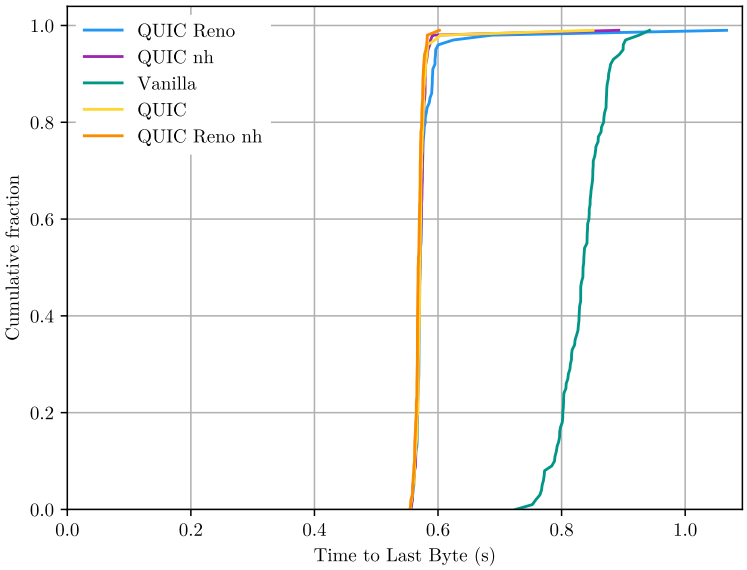

Figure 20: Latency - TTLB 320 KiB 10 ms latency

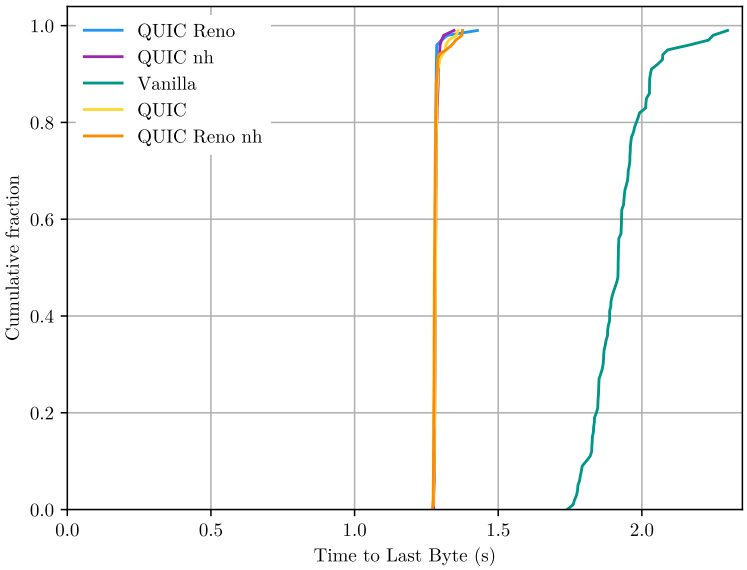

Figure 21: Latency - TTLB 320 KiB 25 ms latency

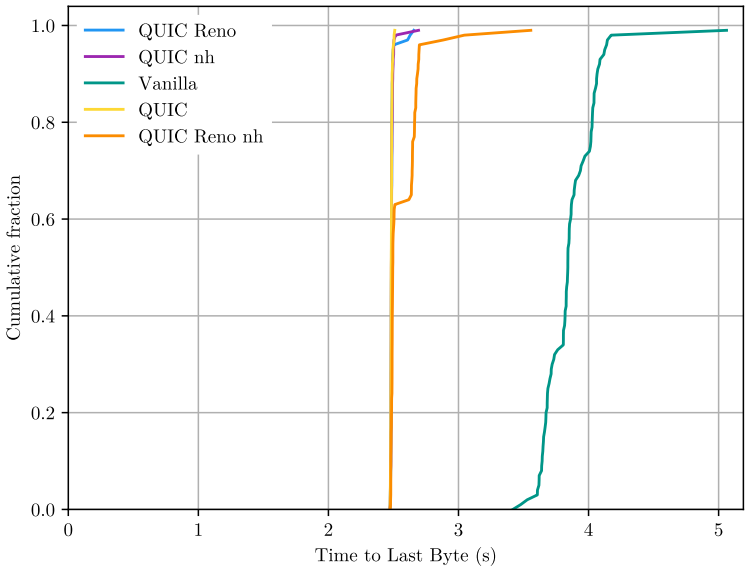

Figure 22: Latency - TTLB 320 KiB 50 ms latency

Table 5: Latency - Average (standard deviation) of TTLB

Latency	10ms	25ms	50ms
QUIC	0.574s (σ 0.029)	1.285s (σ 0.013)	2.484s (σ 0.006)
QUIC nh	0.574s (σ 0.033)	1.284s (σ 0.009)	2.487s (σ 0.022)
QUIC Reno	0.579s (σ 0.052)	1.282s (σ 0.016)	2.489s (σ 0.026)
QUIC Reno nh	0.569s (σ 0.007)	1.282s (σ 0.017)	2.567s (σ 0.143)
Vanilla	0.833s (σ 0.038)	1.919s (σ 0.101)	3.851s (σ 0.205)

In figures 20, 21 and 22, the TTLB values of the 320 KiB download tests are shown for 10 ms, 20 ms and 50 ms latency links. Here, Tor over QUIC both has higher throughput and higher consistency than vanilla Tor. In each of the experiments, on average, vanilla Tor downloads took 40% longer to finish.

In each of the experiments in this scenario, QUIC CUBIC and QUIC Reno perform similarly. This shows that CUBIC does not have a significant advantage over CUBIC, or vice versa, in a simulation without imposed packet loss.

Scenario 3 - Loss

Now, loss is added to the experimental setup. This allows us to explore the performance of both Tor over QUIC and vanilla Tor in an environment with imposed loss. The SLA page of ISP Sprint[51] contains statistics of packet loss throughout the globe. For example, in July 2021, packet loss within Europe

was 0.0018%, and packet loss within the South Pacific was 0.0001%. To find the impact of packet loss on Tor, a packet loss rate of 0.01%, 0.1% and 1% are applied to all links in the network. Bandwidth is again limited to 100 MiB/s and there are no restrictions imposed on latency. QUIC with both CUBIC and Reno, and vanilla Tor are evaluated in this setup. An experiment with 0.001% loss has been conducted as well, but its results are not included as they are not significantly different from the Containernet baseline.

To show the impact of Hystart++ in download performance, test runs with Hystart++ disabled in combination with QUIC CUBIC and QUIC Reno are added to the experiment as well.

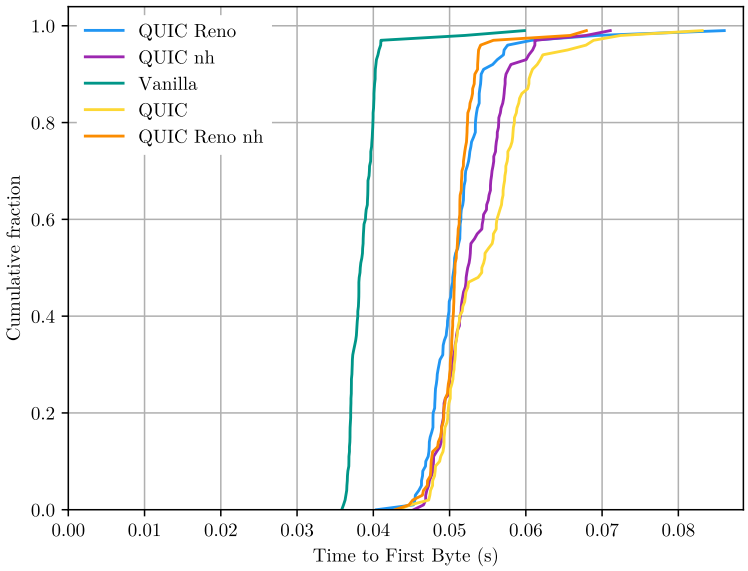

Figure 23: Loss - TTFB 320 KiB, 0.01% loss

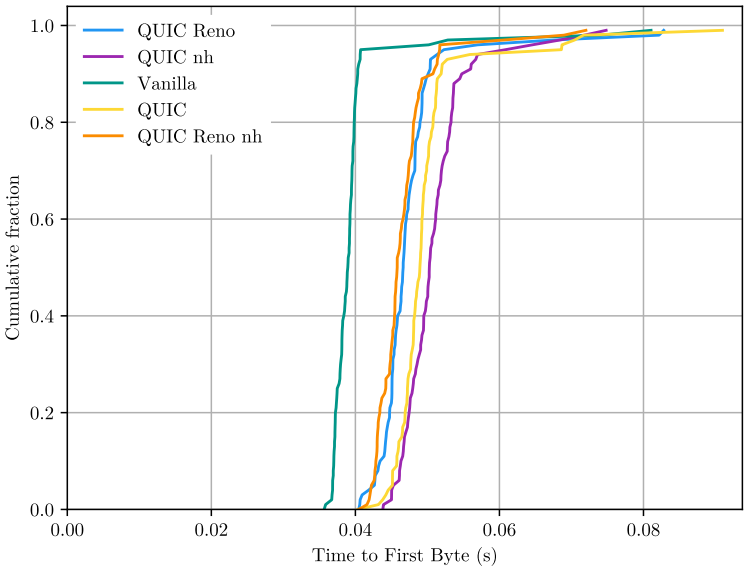

Figure 24: Loss - TTFB 320 KiB, 0.1% loss

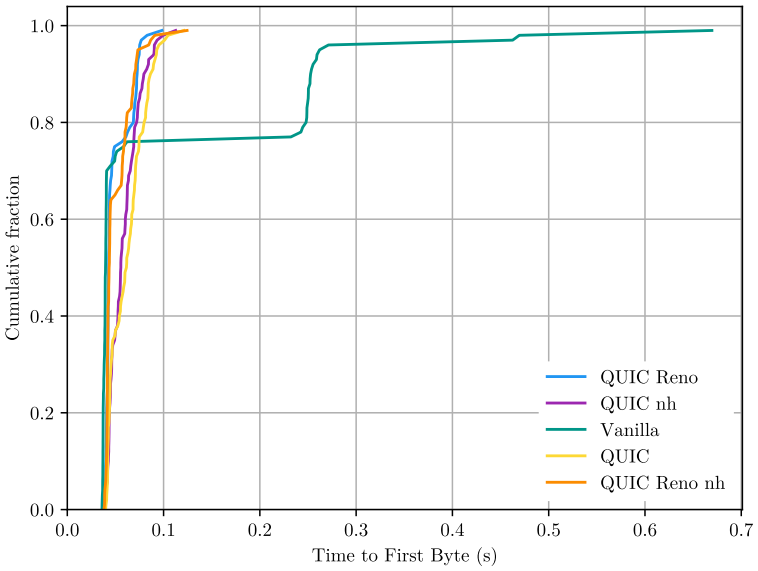

Figure 25: Loss - TTFB 320 KiB, 1% loss

In figure 23 and 24, there is only a small difference in the TTFB values of all

configurations. Similar to figure 15, vanilla Tor's TTFB is lower than any of the QUIC configurations. In figure 25, it can be seen that vanilla Tor's TTFB can be significantly slower than QUIC in 23 of its slowest runs. Where QUIC is resilient against high amounts of packet loss, vanilla Tor depends on timeouts. Where all QUIC configurations finish within 0.13 seconds, vanilla Tor's worst case TTFB took 0.67 seconds.

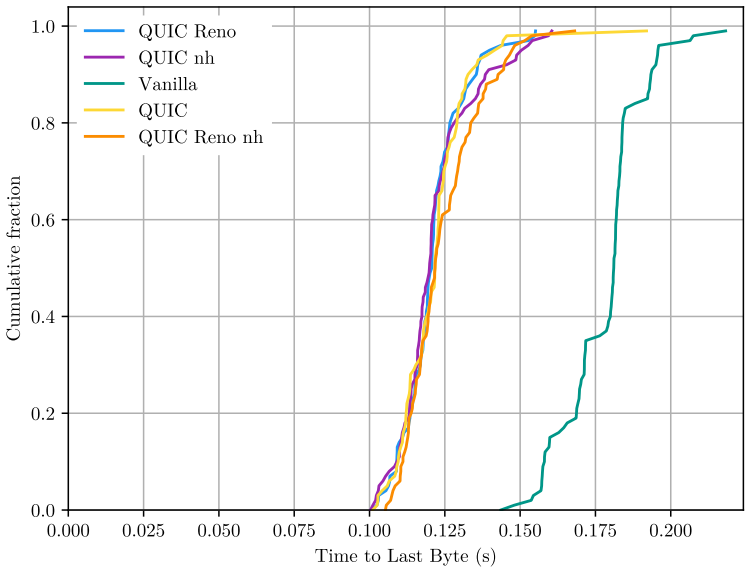

Figure 26: Loss - TTLB 320 KiB, 0.01% loss

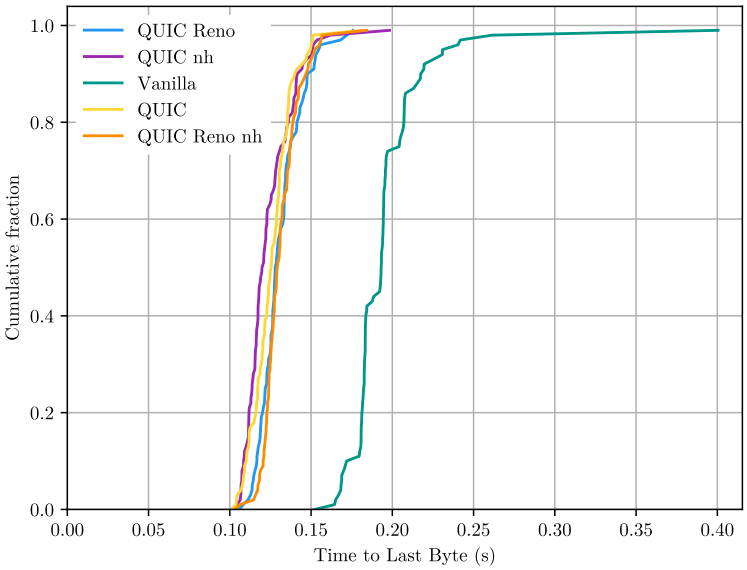

Figure 27: Loss - TTLB 320 KiB, 0.1% loss

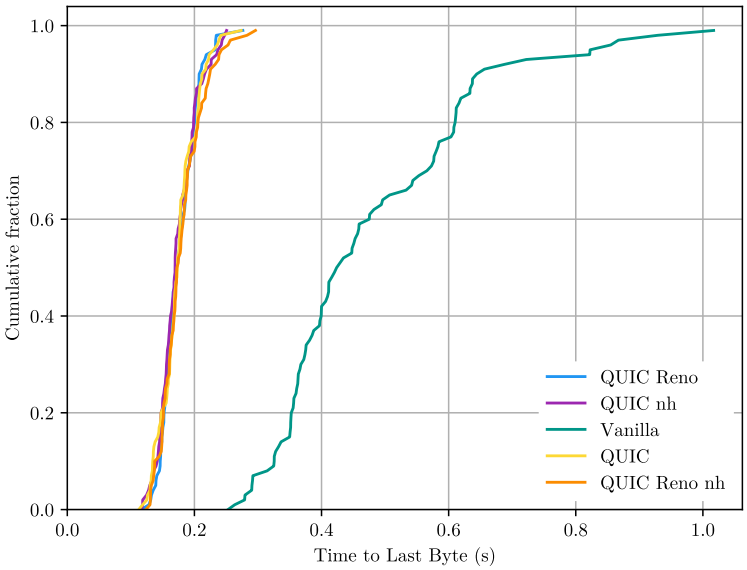

Figure 28: Loss - TTLB 320 KiB, 1% loss

Table 6: Loss - Average (standard deviation) of TTLB

Loss	0.01%	0.1%	1%
QUIC	0.121s (σ 0.012)	0.126s (σ 0.013)	0.175s (σ 0.030)
QUIC nh	0.121s (σ 0.013)	0.124s (σ 0.015)	0.174s (σ 0.030)
QUIC Reno	0.121s (σ 0.010)	0.131s (σ 0.013)	0.176s (σ 0.027)
QUIC Reno nh	0.124s (σ 0.012)	0.131s (σ 0.011)	0.179s (σ 0.034)
Vanilla	0.178s (σ 0.013)	0.195s (σ 0.027)	0.474s (σ 0.155)

In figure 23, 24 and 25 the TTLB measurements are shown. While vanilla Tor generally does better in TTFB performance, vanilla Tor's TTLB falls behind all QUIC configurations in lossy environments. This could be expected as vanilla Tor performed worse than QUIC in the baseline scenario. However, this effect is greater as more loss is added to the environment. The average TTLB for QUIC in this the scenario with 1% loss is 0.175 seconds. In contrast, vanilla's average TTLB is 0.474 seconds, a 171% increase.

In the Containernet baseline scenario, without loss, the average TTLB values within Containernet were 0.123 seconds and 0.178 seconds for Tor over QUIC and vanilla Tor respectively, an increase of 45%. Overall, vanilla Tor is significantly more impacted by loss than Tor over QUIC. For QUIC, the impact of replacing CUBIC with Reno, and the inclusion of Hystart++, are minimal in this scenario. This can be partially explained by the lack of latency in this scenario, which reduces the impact of retransmissions that are not based on timeouts, when the round trip time reaches zero.

Scenario 4 - Jitter

In the original Hystart algorithm, jitter could cause a premature exit from the slow start phase. Hystart++ attempts to remedy this by adding a limited slow start phase (LSS), as described in chapter 2.

On Sprint's SLA webpage[51], it can be seen that the jitter values are generally low. In August 2021, the average measured jitter was 0.1614 ms, 0.0011 ms and 0.4398 ms for North America, Europe, and between Europe to North America respectively.

While the jitter that ISPs measure is relatively low, more jitter will be found in less stable network environments, such as rural areas or in computers connected to Wi-Fi. An experiment with realistic jitter values, below 1 ms, did not yield significantly different results than experiments using no jitter at all. As such, in this scenario, higher jitter values are used than would be found in a healthy network. In later scenarios, a more realistic jitter value of 0.1 ms is used.

In this scenario, experiments with 25 ms latency and jitter values of 10 ms and 25 ms are conducted. No loss is imposed on the network. The jitter in Containernet follows a normal distribution, where the jitter value is the standard deviation of the distribution.

Table 7: Jitter - Average (standard deviation) of TTLB

Jitter	10ms	25ms
QUIC	1.280s (σ 0.012)	1.288s (σ 0.013)
QUIC nh	1.281s (σ 0.014)	1.290s (σ 0.020)
QUIC Reno	1.323s (σ 0.070)	1.708s (σ 0.251)
QUIC Reno nh	1.319s (σ 0.062)	1.909s (σ 0.312)
Vanilla	1.920s (σ 0.093)	1.925s (σ 0.098)

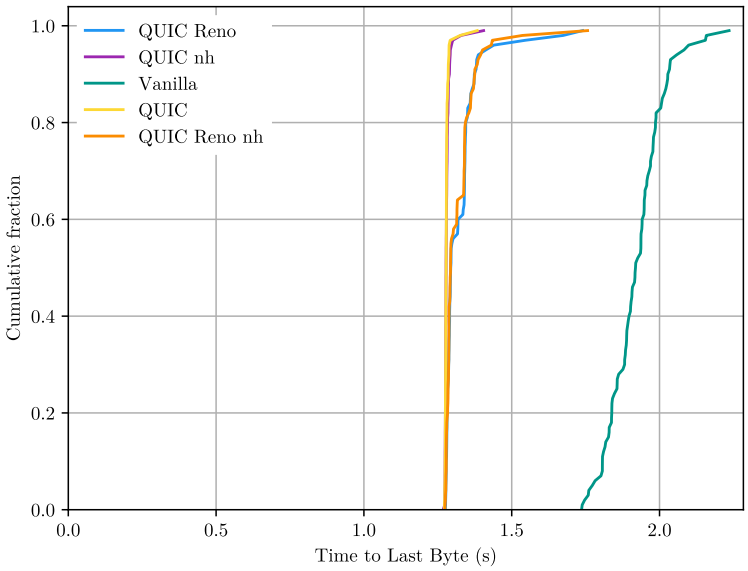

Figure 29: Jitter - TTLB 320 KiB, 25 ms latency, 10 ms jitter

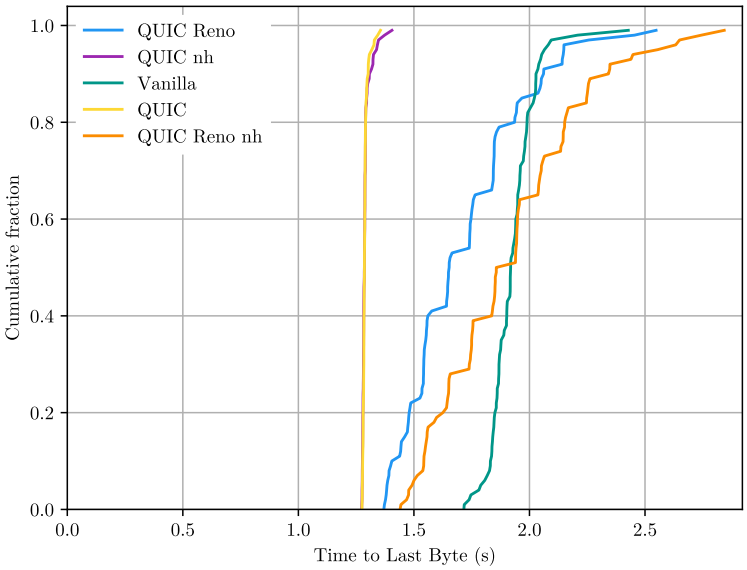

Figure 30: Jitter - TTLB 320 KiB, 25 ms latency, 25 ms jitter

In figure 29 and 30 the results can be seen. The main difference with the experiment with 25 ms latency and no jitter at figure 21, is QUIC with Reno's performance. QUIC with Reno performs roughly 3% worse when 10 ms jitter is added. When 25 ms jitter is added, QUIC with Reno performs 25% worse with Hystart++ enabled, and even 33% worse when Hystart++ is disabled. Contrary to this, both QUIC with CUBIC and vanilla Tor are all within 1% of their measurements with both 10 ms and 25 ms jitter added.

Reno's poor performance can be explained by how it deals with loss. High amounts of jitter cause packet reordering, which can be perceived as loss by congestion control algorithms. CUBIC can quickly return to the congestion window size at which loss occurred due to its 30% window reduction and concave growth. Reno will reduce its congestion window by 50% when loss is detected, and it will only increase its window linearly in congestion avoidance. This keeps Reno's congestion window low in high jitter network environments.

Scenario 5 - Realistic network

Next, the network restrictions of the previous scenarios are combined to create a realistic network environment.

In this scenario, a realistic network environment is simulated to explore Tor's performance. As previously substantiated, a latency value of 25 ms is used, 0.01% loss is added, and the per-link bandwidth is capped at 100 MiB/s. Additionally, 0.1 ms jitter is added, as described in scenario 4.

Table 8: Realistic - Average (standard deviation) of TTLB

Realistic network	
QUIC	1.289s (σ 0.032)
QUIC nh	1.308s (σ 0.050)
QUIC Reno	1.402s (σ 0.083)
QUIC Reno nh	1.403s (σ 0.085)
Vanilla	1.968s (σ 0.175)

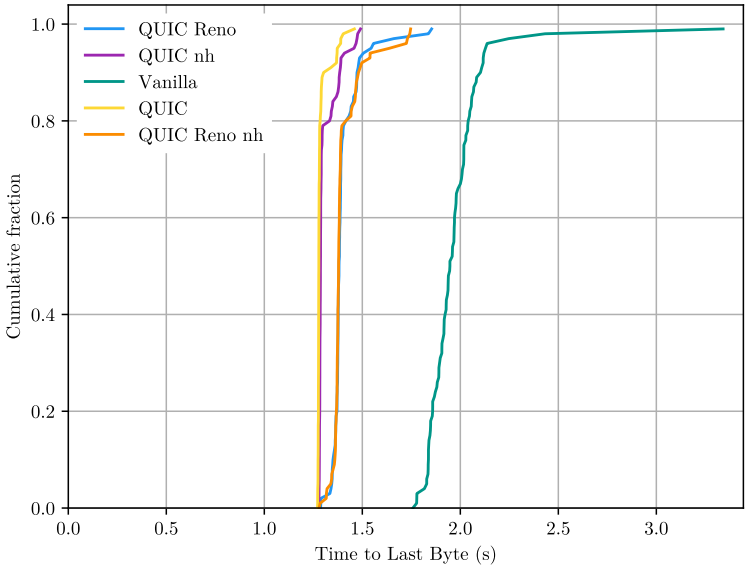

Figure 31: Realistic network - TTLB 320 KiB, 25 ms latency, 0.01% loss, 0.1 ms jitter

In previous experiments QUIC and QUIC without Hystart++ performed similarly. As seen in figure 31, in an experiment which combines latency, jitter and loss, Hystart++ makes more of an impact. Especially in the 20% slowest runs, QUIC with Hystart++ outperforms QUIC without Hystart++. This can be attributed to Hystart++'s ability to converge faster to a stable window size, and Hystart++'s limited slow start phase, as explained in Chapter 2.

As with previous experiments, QUIC vastly outperforms vanilla Tor in this single-circuit setup.

Scenario 6 - Fairness

To measure the fairness between multiple circuits, multiple clients will download through the Tor network simultaneously. Each circuit's path is preconfigured,

such that channels do not overlap at random. Instead, all circuits will share exactly one channel between the second hop and the exit node.

To calculate the fairness of throughput, Jain's fairness index is used[30]. Jain's index is commonly used to compute fairness in networks. The index works for any number of connections, is independent of scale, and, unlike min-max fairness, Jain's fairness index is continuous. It is defined as such:

$$F = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} {x_i}^2}$$

In this formula, n is the number of connections, x_i is the throughput of connection i, and F is the fairness, which is between $\frac{1}{n}$ and 1.

For this experiment, two separate clients will download 5000 KiB through Bwtool simultaneously. The environment is identical to the one used in the realistic network scenario. The two circuits will have the same middle node and exit node, but distinct clients and entry nodes. This means that exactly one channel is shared between both circuits. Five runs are done for both QUIC with CUBIC and vanilla Tor.

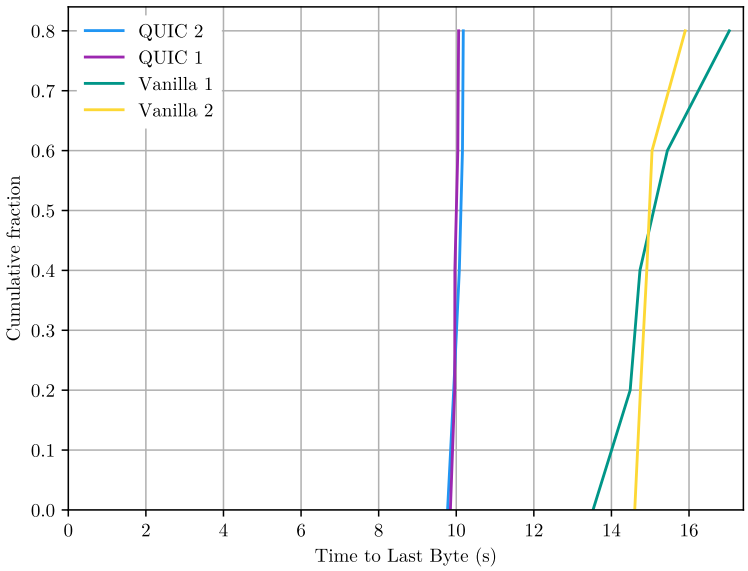

Figure 32: Fairness - 2 simultaneous clients, TTLB 5000 KiB, 25 ms latency, 0.01% loss, 0.1 ms jitter

Table 9: Fairness - Average (standard deviation) of bandwidth and Jain's fairness.

	Bandwidth (KB/s)	Jain's fairness
QUIC	512 (σ 6.64)	0.99996 (σ 2.75e-5)
Vanilla	341 (σ 20.6)	0.998 (σ 3.04e-3)

In figure 32, performance of each individual client can be seen. In table 9, the average bandwidth and the average values of Jain's fairness index are shown. In this experiment, QUIC features both a higher throughput and near perfect fairness between clients in each of the runs. This satisfies the initial goal, which aims for the QUIC implementation to not concede fairness to vanilla Tor.

Scenario 7 - Load

For the last scenario, the performance of Tor over QUIC is evaluated in a larger scale network. This scenario includes 16 relay nodes, 16 client nodes and 3 directory authorities. 15 client nodes will download 5000 KiB repeatedly to add load to the network. The last client is used to download 320 KiB, which is measured. In this scenario, vanilla Tor and Tor over QUIC are evaluated, along with each combination of Reno and Hystart++. The same network restriction as in scenario 5, are used, such that this scenario may resemble real world performance.

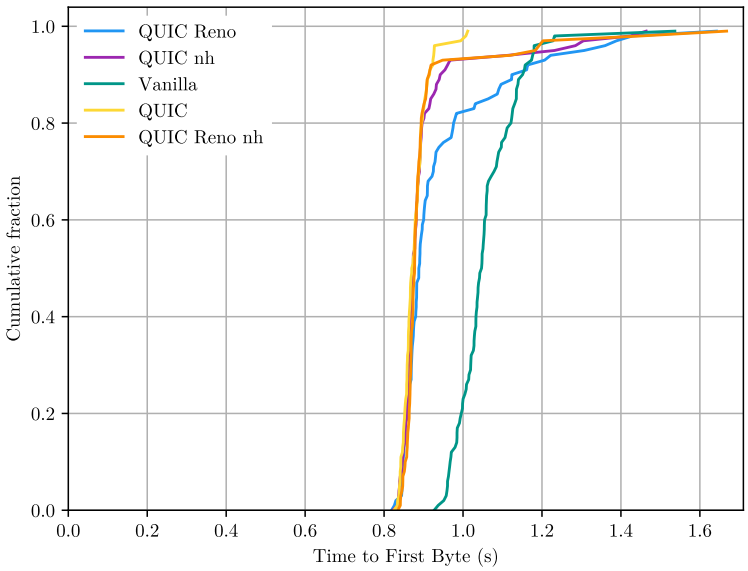

Figure 33: Load - 15 continuous downloaders, TTFB 320 KiB, 25 ms latency, 0.01% loss, 0.1 ms jitter

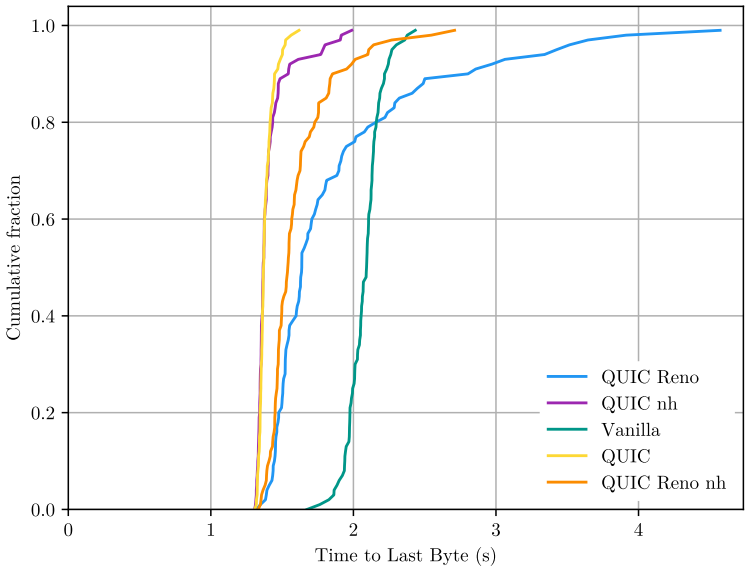

Figure 34: Load - 15 continuous downloaders, TTLB 320 KiB, 25 ms latency, 0.01% loss, 0.1 ms jitter

Table 10: Load - Average (standard deviation) of TTLB

Load	
QUIC	1.384s (σ 0.054)
QUIC nh	1.409s (σ 0.129)
QUIC Reno	1.868s (σ 0.606)
QUIC Reno nh	1.601s (σ 0.237)
Vanilla	2.076s (σ 0.122)

As shown in figure 33, on average, all QUIC variants outperform vanilla Tor in TTFB performance. Additionally, QUIC with CUBIC and Hystart++ outperforms all other QUIC variants, especially in the 10% slowest measurements. Vanilla Tor's 99th percentile performance is on par with all QUIC variants but QUIC with CUBIC and Hystart++. This inspires confidence in the performance characteristics of combining QUIC with CUBIC and Hystart++. Contrary to this, QUIC with Reno performs significantly worse when Hystart++ is enabled than when it is not. This effect is apparent in figure 34 as well, which shows TTLB performance. Where QUIC with CUBIC benefits from Hystart++, especially in its slower runs, QUIC with Reno performs significantly worse when Hystart++ is enabled.

Generally, in this scenario, QUIC with CUBIC is superior to QUIC with Reno. QUIC with CUBIC and Hystart++ performs better than any other QUIC variant.

Additionally, all QUIC variants generally outperform vanilla Tor.

On average, QUIC with CUBIC and Hystart++ features 50% better download performance than vanilla Tor within this loaded test network.

6. Conclusion

In this thesis, the integration of QUIC version 1 within Tor has been designed and implemented. Containernet, a flexible, Docker-based, network test bed has been used to test this implementation in various network environments.

By mapping Tor circuits to QUIC streams, our implementation eliminates the head-of-line blocking problem that is present in vanilla Tor. This lets users achieve higher bandwidth within the Tor network when loss occurs.

Our implementation is designed with deployability in mind. This is reflected in the choice of QUIC library, Quiche, which implements version 1 of the QUIC protocol. QUIC is more fit for production use than previously proposed alternative transport protocols because it is built on top of UDP, and because of its exhaustive encryption which prevents ossification. Additionally, because QUIC implemented as a user space protocol, it can be bundled with Tor without requiring kernel modifications.

Due to the hop-by-hop nature of the implementation, replacing TCP with QUIC has a limited impact on the security and anonymity of the Tor network, though more research is to be done on this topic.

Our implementation is evaluated through Containernet, a flexible Docker-based network test bed. Containernet allows the implementation to be evaluated in a wide range of network environments. Additionally, because both the source code of the Tor modifications and the test bed are open source, the results of this thesis are easily reproduced, such that future work can build on our work.

It is shown that QUIC over Tor performs best when used with CUBIC and Hystart++ enabled, rather than when using the Reno congestion control algorithm. Additionally, QUIC over Tor, when used with CUBIC and Hystart++, outperforms vanilla Tor in all evaluated environments. QUIC over Tor has a larger advantage over Tor when more latency and loss is added. However, in a scenario with realistic network conditions and high load on the Tor network, Tor over QUIC still shows a 50% performance improvement over vanilla Tor. Besides providing an overall higher bandwidth to its users, Tor over QUIC distributes its bandwidth more fairly amongst its users, reaching a near perfect score on Jain's fairness index.

Even though the QUIC protocol and Quiche are production ready, the code as delivered with this thesis needs more work before it can be included with Tor.

In its current state, Quiche does not expose the certificates that are used for

its handshake. Meanwhile, the Tor link handshake as it is implemented in TLS channel uses the certificates of its TLS connection to establish authentication on top of TLS. As such, the current QUIC implementation simply sends its "authenticated" identity over the wire. This is not secure, and the Tor link handshake implementation needs more work to be made secure again. Since this thesis focuses on performance, rather than security and its attack vectors, future work could fix this.

Within this study, QUIC has been shown to perform well within network with up to 15 Tor relays. To get a better sense of the performance of Tor in production, a larger scale experiment needs to be performed. It would be ideal if the Shadow simulator were to be adapted as to support the UDP based QUIC protocol. Since Shadow has a focus on relatively large experiments on Tor, this would be a good fit for exploring Tor over QUIC's properties in such an environment.

Right now, the KIST scheduler cannot be used within Tor over QUIC as-is. This is because KIST uses kernel information about used TCP sockets to make an informed decision about optimal scheduling. QUIC uses UDP such that KIST's fallback, KISTLite, is used for scheduling when QUIC is enabled.

Additionally, the interface of the scheduler does not account for the possibility of scheduling based on differences of individual streams. QUIC multiplexes different circuits over the same QUIC connection, and QUIC allows for stream-specific congestion and flow control. It can be beneficial for the scheduler to access the capacity that QUIC allows each stream to send. Through this, QUIC streams that are limited by flow or congestion control would not receive cells from the scheduler. On the other hand, QUIC streams that have more available capacity could receive more cells from the scheduler.

Since this thesis is mainly focused on performance, more work can be done to explore the security considerations of using Tor over QUIC. The traffic characteristics of QUIC are different from the characteristics of TCP + TLS. Research can be done to investigate whether this allows for fingerprinting.

Additionally, because QUIC's multiplexing is more efficient, it's possible that end-to-end traffic looks less similar than in vanilla Tor. Research can be done to see whether it's possible to create a covert channel based on delayed cells on one end, and then perceiving this delay on the other end of the circuit.

Even though QUIC's hop-by-hop congestion control is shown to be efficient, Tor's basic end-to-end flow control window can still lead to performance deficiency. It is possible to signal congestion up the circuit through backpressure, such that the sender can limit its sending rate. QUIC is ideal for this scenario, as it comes with per-stream rate limiting out of the box. Right now, backpressure is not utilized, and it is still possible for relays to become congested, adding latency for all circuits that pass through. Backpressure based flow control can help this problem.

QUIC is now implemented at the channel level, rather than at the connection level. This makes it easier to switch between these two implementations, as both follow the same channel interface. The TLS channel and QUIC channel classes can be made to coexist without too much difficulty. However, ideally a connection will eventually default to QUIC, and fallback to TLS if QUIC is

not supported. This fallback mechanism needs to be implemented. When this is done, QUIC could be rolled out to Tor gradually. As such, when two relays connect, they can already provide the benefits that QUIC has to offer without requiring QUIC support from the other relays on the circuit.

Overall, QUIC's performance within Tor is very promising, and few hurdles remain for QUIC to be integrated into Tor.

References

[1] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," Naval Research Lab Washington DC, 2004.

[2] A. Forte, N. Andalibi, and R. Greenstadt, "Privacy, anonymity, and perceived risk in open collaboration: A study of tor users and wikipedians," in *Proceedings of the 2017 ACM conference on computer supported cooperative work and social computing*, 2017, pp. 1800–1811.

[3] "Tor metrics." https://metrics.torproject.org/.

[4] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport." RFC 9000; RFC Editor, May-2021.

[5] F. Tschorsch and B. Scheuermann, "Mind the gap: Towards a backpressure-based transport protocol for the tor network," in *13th USENIX symposium on networked systems design and implementation (NSDI 16)*, 2016, pp. 597–610.

[6] C. Viecco, "UDP-OR: A fair onion transport design," HotPETS, 2008.

[7] J. Reardon and I. Goldberg, "Improving tor using a TCP-over-DTLS tunnel," in *Proceedings of the 18th conference on USENIX security symposium*, 2009, pp. 119–134.

[8] M. F. Nowlan, D. I. Wolinsky, and B. Ford, "Reducing latency in tor circuits with unordered delivery," in *3rd USENIX workshop on free and open communications on the internet (FOCI 13)*, 2013.

[9] A. Clark, "QUUX: a QUIC un-multiplexing of the Tor relay transport," Master's thesis, University College London, United Kingdom, 2016.

[10] W. Sabée, "Adding QUIC support to the Tor network," Master's thesis, TU Delft, The Netherlands, 2003.

[11] M. Peuster, H. Karl, and S. van Rossem, "MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments," in *2016 IEEE conference on network function virtualization and software defined networks (NFV-SDN)*, 2016, pp. 148–153.

[12] V. Tyagi, S. Pandey, and T. Kumar, "A survey of TCP congestion control algorithm in wireless network: BIC and CUBIC," in *International conference on technological and management advances in the new age economy: An industry perspective*, 2014.

[13] J. Choi, "CUBIC and HyStart++ support in quiche," May-2020. [Online]. Available: https://blog.cloudflare.com/cubic-and-hystart-support-in-quiche/.

[14] P. Balasubramanian, Y. Huang, and M. Olson, "HyStart++: Modified Slow Start for TCP," Internet Engineering Task Force; Internet Engineering Task Force, Internet-Draft draft-ietf-tcpm-hystartplusplus-03, Jul. 2021.

[15] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3." RFC 8446; RFC Editor, Aug-2018.

[16] B. Hesmans, F. Duchene, C. Paasch, G. Detal, and O. Bonaventure, "Are TCP extensions middlebox-proof?" in *Proceedings of the 2013 workshop on hot topics in middleboxes and network function virtualization*, 2013, pp. 37–42.

[17] N. Sullivan, "Why TLS 1.3 isn't in browsers yet," *The Cloudflare Blog*, Aug-2018. [Online]. Available: https://blog.cloudflare.com/why-tls-1-3-isnt-in-browsers-yet/.

[18] J. Corbet, "QUIC as a solution to protocol ossification," *[LWN.net]*, Jan-2018. [Online]. Available: https://lwn.net/Articles/745590/.

[19] J. Roskind, "QUIC - MULTIPLEXED STREAM TRANSPORT OVER UDP," 2012. [Online]. Available: https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit.

[20] "Experimenting with QUIC," *Chromium Blog*, Jun-2013. [Online]. Available: https://blog.chromium.org/2013/06/experimenting-with-quic.html.

[21] J. Iyengar and I. Swett, "QUIC Loss Detection and Congestion Control." RFC 9002; RFC Editor, May-2021.

[22] S. Floyd, J. Mahdavi, M. Mathis, and Dr. A. Romanow, "TCP Selective Acknowledgment Options." RFC 2018; RFC Editor, Oct-1996.

[23] I. Poese, S. Uhlig, M. A. Kaafar, B. Donnet, and B. Gueye, "IP geolocation databases: unreliable?" *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 2, pp. 53–56, Apr. 2011.

[24] "Tor's source code," *tor*. [Online]. Available: https://gitweb.torproject.org/tor.git/.

[25] "Tor's protocol specifications," *tor*. [Online]. Available: https://gitweb.torproject.org/torspec.git/.

[26] R. Dingledine and S. J. Murdoch, "Performance improvements on tor or, why tor is slow and what we're going to do about it," 2009. [Online]. Available: https://svn-archive.torproject.org/svn/projects/roadmaps/2009-03-11-performance.pdf.

[27] R. Jansen, M. Traudt, J. Geddes, C. Wacek, M. Sherr, and P. Syverson, "KIST: Kernel-informed socket transport for tor," *ACM Trans. Priv. Secur.*, vol. 22, no. 1, Dec. 2018.

[28] M. AlSabah, K. Bauer, I. Goldberg, D. Grunwald, D. McCoy, S. Savage, and G. M. Voelker, "DefenestraTor: Throwing out windows in tor," in *Privacy enhancing technologies*, 2011, pp. 134–154.

[29] "Chrome's QUIC implementation." [Online]. Available: https://chromium.googlesource.com/chromium/src/net/+/refs/heads/main/quic.

[30] R. K. Jain, D.-M. W. Chiu, W. R. Hawe, and others, "A quantitative measure of fairness and discrimination," *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 1984.

[31] Jaapp-, "Jaapp-/tor: Tor over QUIC," *GitHub*. [Online]. Available: https://github.com/Jaapp-/tor.

[32] J. Rüth, I. Poese, C. Dietzel, and O. Hohlfeld, "A first look at QUIC in the wild," in *Passive and active measurement*, 2018, pp. 255–268.

[33] K. Hogan, "Security analysis of Tor over QUIC," Master's thesis, MIT, 2020.

[34] "QUICHE," *Google Git*. [Online]. Available: https://quiche.googlesource.com/quiche.

[35] Facebookincubator, "Facebookincubator/mvfst: An implementation of the QUIC transport protocol." *GitHub*. [Online]. Available: https://github.com/facebookincubator/mvfst.

[36] ngtcp2, "ngtcp2/ngtcp2: ngtcp2 project is an effort to implement IETF QUIC protocol," *GitHub*. [Online]. Available: https://github.com/ngtcp2/ngtcp2.

[37] Quinn-Rs, "Quinn-rs/quinn: Futures-based QUIC implementation in rust," *GitHub*. [Online]. Available: https://github.com/quinn-rs/quinn.

[38] Mozilla, "Mozilla/neqo," *GitHub*. [Online]. Available: https://github.com/mozilla/neqo.

[39] Cloudflare, "Cloudflare/quiche: Savoury implementation of the QUIC transport protocol and HTTP/3," *GitHub*. [Online]. Available: https://github.com/cloudflare/quiche.

[40] Inc. OpenSSL Foundation, "OpenSSL." [Online]. Available: https://www.openssl.org/.

[41] "BoringSSL." [Online]. Available: https://boringssl.googlesource.com/boringssl/.

[42] R. Jansen and N. Hopper, "Shadow: Running tor in a box for accurate and efficient experimentation," in *Proceedings of the 19th symposium on network and distributed system security (NDSS)*, 2012.

[43] Torproject, "Torproject/chutney: Unofficial git repo – report bugs/issues/pull requests on https://gitlab.torproject.org/ –," *GitHub*. [Online]. Available: https://github.com/torproject/chutney.

[44] M. P. Contributors, *Mininet*. [Online]. Available: http://mininet.org/.

[45] Jaapp-, "Jaapp-/tor-containernet," *GitHub*. [Online]. Available: https://github.com/Jaapp-/tor-containernet.

[46] "Methodically modeling the tor network," in *5th workshop on cyber security experimentation and test (CSET 12)*, 2012.

[47] arma, "Bittorrent over tor isn't a good idea," *Tor Blog*, Apr-2010. [Online]. Available: https://blog.torproject.org/bittorrent-over-tor-isnt-good-idea.

[48] "Advanced usage," *Advanced Usage - Requests 2.26.0 documentation*. [Online]. Available: https://docs.python-requests.org/en/master/user/advanced/.

[49] "Http.client - HTTP protocol client," *http.client - HTTP protocol client - Python 3.9.6 documentation*. [Online]. Available: https://docs.python.org/3/library/http.client.html.

[50] "Performance | tor metrics," *Tor Metrics*. [Online]. Available: https://metrics.torproject.org/onionperf-latencies.html.

[51] *IP/MPLS Products from Sprint*. [Online]. Available: https://www.sprint.net/tools/sla-performance/sl.