Adding QUIC support to the Tor network

W. F. Sabée

Delft University of Technology

Adding QUIC support to the Tor network

Master's Thesis in Embedded Systems

Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

W. F. Sabée

Tuesday 27th August, 2019

Author
W. F. Sabée

Title
Adding QUIC support to the Tor network

MSc presentation
Friday 30th August, 2019

Graduation Committee

Dr.ir. J.A. Pouwelse	Delft University of Technology
Dr. S. Roos	Delft University of Technology
Dr. N. Yorke-Smith	Delft University of Technology

Abstract

Privacy in the Internet is under attack by governments and companies indiscriminately spying on everyone. The anonymity network Tor is a solution to restore some privacy, however, Tor is slow in both bandwidth and latency. It uses a TCP-based connection to multiplex different circuits between nodes and this causes different independent circuits to interfere with each other. To solve this, we propose a transport layer implementation using the UDP-based protocol QUIC, as it allows independent streams over a single connection. We built a Tor prototype that uses this protocol and evaluated its performance using a custom network simulator, as existing simulators were shown to be incompatible. We show that the QUIC-based implementation increased performance in several of the use case scenarios, mainly outperforming on the 'time to first byte' metric. However, due to certain analysis issues, not all results are conclusive and continued work on our prototype is required and encouraged.

Preface

> No one shall be subjected to arbitrary interference with his privacy, family, home or correspondence, nor to attacks upon his honor and reputation. Everyone has the right to the protection of the law against such interference or attacks.
>
> *Article 12 of the 1948 Universal Declaration of Human Rights*

Thanks to my supervisor Stefanie, who helped me get through it with her advice and weekly meetings, to all the proofreaders for their very valuable feedback, with in particular Kian, and every one else who supported me, even when at times I had to tell them no for being too busy.

W. F. Sabée

Delft, The Netherlands
Tuesday 27th August, 2019

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Privacy and anonymity on the Internet are all but an illusion. Edward Snowden gave us proof that governments all around the world are indiscriminately watching our every move online, with government intelligence agencies such as the NSA adopting the all-telling internal slogan 'Collect it all' [20]. And this is not without consequences. The Freedom on the Net 2018 report estimates that 71% of all Internet users live in a country where Internet users have either been arrested or imprisoned for the political, social or religious content they posted on the Internet [23]. And due to the underlying architecture of the protocols upon which the Internet is built, anything you do online can be linked back to you if no special precautions are taken.

It is not just governments that Internet users have to fear are spying on them. In 2017, The Economist reported that data had overtaken oil as the worlds most valuable resource [11]. This has led to what some call the age of Surveillance Capitalism, where companies such as Google and Facebook have their entire business models built around collecting as much data as possible about as many people as possible, all in the name of targetting those people with advertisements. And often breaking the law in doing so, even after warnings or sometimes record breaking fines [2, 3, 16, 24, 40].

To illustrate to which extend these companies have access to your online activity, Englehardt et al. found that about 85% of the websites in the Alexa top 1 million site list will cause your web browser to make contact with Google servers on visit, while about 35% of the sites in that list will make your web browser connect to Facebook servers in the background [12]. And this practice has only been increasing over the years [5, 33]. Even worse, Maris et al. found that 93% of pornography sites, arguably one of the most private activities that people conduct on the web, leak information about their users to a third party [37]. Especially in countries where some sexual orientations or activities are prohibited by law or can even carry a capital punishment, collecting and sharing this kind of sensitive data with third parties can be life ruining.

Fortunately, there are precautions users can take to increase their privacy and

stay anonymous on the Internet. One of them is the usage of the Tor network to mask the user's IP address [9]. Tor is a free anonymity network that is run by volunteers and consists of several thousands of Tor nodes [48]. The intended use of Tor is to provide its users privacy by preventing network surveillance by for example governments or network providers. Simultaneously it provides its users with anonymity from who they are communicating with, such as the website they are visiting. It does so by routing the user's traffic through a number of Tor nodes, as explained in more detail in Chapter 2, with Figure 2.1 in particular. A network provider can only observe that the user is using Tor, but not for which purpose. The visited website can only tell the visitor is using Tor, but not who the user is. Under the assumption that an adversary can not observe the whole Tor network or control a majority of the Tor network, the network traffic of the user can not be directly linked back to them.

1.1 Problem description

Tor is slow and this is a problem [10]. That users value speed can be illustrated by two statistics: Amazon said an increase in load time of its webpage with 100 ms resulted in a drop in sales of 1% [31], while Google discovered that increase in page load time from 0.4 seconds to 0.9 seconds decreased traffic and ad revenues by 20% [34].

While Tor will always inherently be slower than normal, identifiable Internet usage because of its design (see Chapter 2), part of the performance issues can be attributed to the way Tor handles traffic between nodes: all traffic between two nodes is multiplexed over one single TCP connection. For Tor circuits that share the same TCP connection, this introduces two notable problems:

- **Head-of-line blocking**: Since TCP traffic is delivered in order and reliable, dropped packets block all other packets on the same connection until the missing packet is retransmitted. One dropped packet will therefore increase the latency of the packets directly after it as they wait for the dropped packet to be retransmitted, even if these waiting packets have already been received by the receiver and are unrelated to the dropped packet.

- **TCP congestion control**: The second problem is caused by the way TCP controls congestion. Whenever too many packets are dropped, the congestion window is reduced by 50%. When this is caused by a high-latency high-bandwidth circuit, any unrelated low-latency low-bandwidth circuits will unfairly suffer from the increased latency because they belong to the same TCP connection.

These problems have been known for over a decade and one proposal is to use a UDP-based connection instead [10].

1.2 Research questions

The research question this thesis aims to answer is the following:

> What are the effects on the performance of the Tor network when using a UDP-based protocol as the transport layer protocol?

To answer the general research question, the following subquestions will be evaluated:

- How can one best integrate a new transport layer protocol into Tor?
- How can one measure the performance impact of using another transport layer protocol in Tor?
- What are the performance impacts of using a UDP-based protocol as transport layer protocol on the head-of-line blocking problem in a simple best or worst case scenario?
- What are the performance impacts of using a UDP-based protocol as transport layer protocol on the TCP congestion control problem in a simple best or worst case scenario?
- What are the performance impacts in a combined scenario modelled after the real Tor network of using a UDP-based protocol as transport layer protocol?

1.3 Contribution

This thesis builds upon the existing hypothesis that the TCP-based protocol that is used by Tor to communicate between nodes has a significant impact on why Tor is slow and that a UDP-based transport protocol could improve this. It provides arguments as to why the QUIC transport protocol is a good candidate for this UDP-based protocol and details a design based on a list of requirements that are formulated in Section 5.1. To evaluate the performance of this design, the metrics on how the performance can be measured are defined and a prototype implementing the design is created. Because existing network simulation tools were not sufficient in this case, a script was developed to run the Tor network inside a virtualized environment and existing tools were modified to utilize this environment and to keep a log of the metrics that are used to evaluate the performance.

1.4 Thesis outline

The contents of this document are structured as follows: Chapter 1 start with the introduction, an overview of the research question and a problem description. Chapter 2 gives background about the Tor network, while Chapter 3 gives background

about various Internet protocols. Next, Chapter 4 gives an overview of related work. Chapter 5 sets out the design of how to implement QUIC in Tor, while Chapter 6 documents the actual implementation. In Chapter 7 methods to simulate the Tor network are detailed and in Chapter 8 the implementation is evaluated using the chosen method. Finally, Chapter 10 draws a conclusion from the results and Chapter 9 lists several future improvements.

Chapter 2

The Tor network

The Tor network is a circuit-based, low latency anonymity network designed to anonymize TCP traffic, like web browsing and messaging. The nodes that participate in the Tor network are publicly known servers that are run by volunteers that donate their bandwidth to the network. To use the Tor network, clients create a circuit of nodes in which the nodes only know their predecessor and their successor, but not any other node in the circuit [9]. Traffic is then relayed between those nodes and because none of them know the full circuit, this breaks linkability between the Tor client on one side of the circuit, and the traffic destination on the other side of the circuit. A simplified view of the make-up of a Tor connection is shown in Figure 2.1, where the user's Tor client uses an encrypted connection via three randomly picked Tor nodes (called a circuit) to communicate with the web.

Tor's threat model makes the assumption that an adversary is not able to observe the whole network, also known as a global passive adversary. Because of the low-latency nature of the Tor network, such a strong adversary would be able to follow the traffic all the way through the network from node to node and link the Tor client and the destination of the traffic. A high-latency anonymity network where nodes collect traffic, hold it, mix it with other traffic and only then relay it in batches defends against this, but this adds such a significant delay to each hop (think tens of seconds, minutes or more) that this makes it unsuitable for low-latency applications such as browsing the web. Instead, Tor assumes that an adversary is only able to observe some subset of the network and in addition is able to generate, modify, delete or delay traffic, for example by operating or compromising some of the nodes.

Tor's threat model also makes the assumption that there is no single adversary that is in control of a large portion of the Tor nodes. An adversary would need to grow its part in the network slowly, in different jurisdictions and with different characteristics as to not cause any suspicion. And even then it is a game of chance for the Tor client to pick a circuit with enough related nodes. Less sophisticated adversaries that add a large number of nodes in a short timespan without explicitly declaring they are related (so that the Tor client knows not to combine them) are

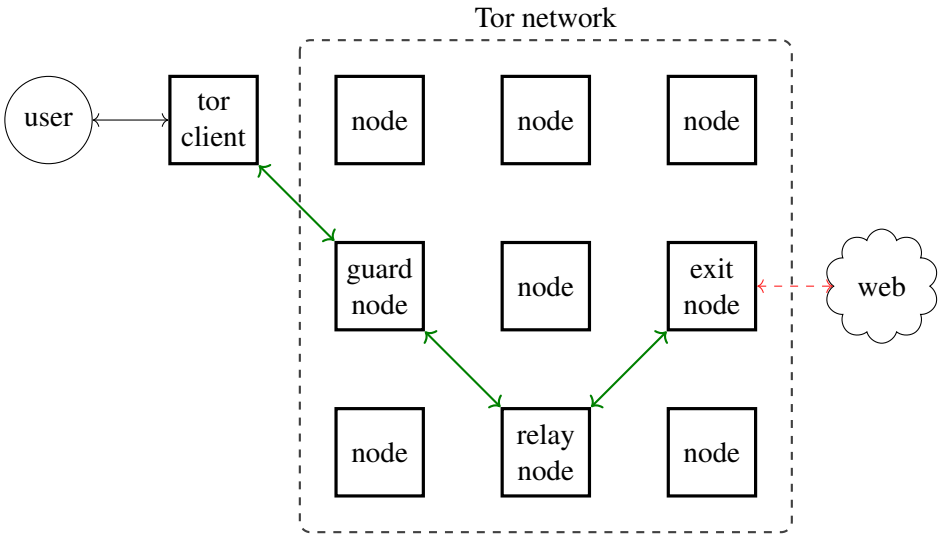


Figure 2.1: A schematic view of a circuit in the Tor network. The user's Tor client randomly picks a guard node, a relay node and an exit node to set up an encrypted connection (the green arrows) to anonymously communicate with a web site. Only the client knows of which nodes the full circuit consists of, while the participating nodes only know their predecessor and their successor in the circuit, which breaks linkability.

regularly detected and removed [43].

In the following sections, each layer of the internal and network design of the Tor network will be detailed. A schematic view of the different layers can be found in Figure 2.2.

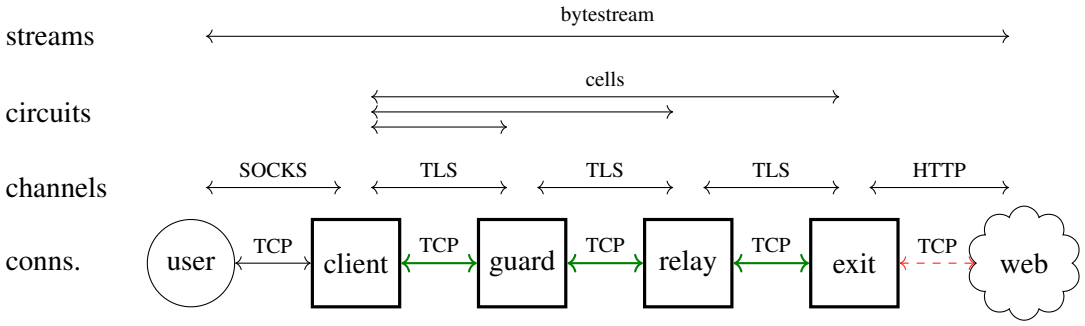


Figure 2.2: A schematic view of the network layers in a Tor circuit, such as the one shown in Figure 2.1. On the bottom layer are TCP connections (Section 2.1). Connections denoted by a black arrow are local connections, a green arrow connections with encrypted data, while the red arrow denotes connections with unencrypted data. The user uses the SOCKS protocol to talk to the Tor client, while the Tor nodes talk over TLS channels (Section 2.2). Over these channels, the client sends cells (Section 2.5) to build a circuit (Section 2.3), step by step. Over this circuit a TCP-like stream (Section 2.4) is established from the user to the destination. The last node decrypts the data (in this case, HTTP) and forwards it to the destination.

2.1 Connections

The most fundamental layer that Tor uses to communicate between nodes in the network are connections. The connections between Tor nodes are TCP based (see Section 3.1) and are used to transport an ordered and reliable stream of bytes between the nodes. Between every node there is only one connection, which means that the circuits (see Section 2.3) of multiple clients are multiplexed over a single connection. This is done for both performance and anonymity reasons. Reusing an existing connection means no time is spent on a TCP handshake every time a new circuit is built. In addition to that, heavily used relays might serve tens or hundreds of thousands of nodes at the same time. Keeping open that many TCP connections uses up unnecessary resources and might not even be possible by default on some operating systems. Reusing connections also means that an outside passive observer can not easily distinguish between the traffic of different circuits, improving anonymity and the unlinkability of those circuits.

2.2 Channels

Channels are an abstraction over the connections between Tor nodes that are used to carry cells (see Section 2.5). They are mostly an internal abstraction of Tor that is used to add encryption to the connection and to provide a simpler cell based interface between nodes. Currently, the only channel implementation in Tor is the TLS channel, that creates a TLS connection between nodes and buffers byte

streams to send and receive cells.

The TLS encryption that the channels utilize, uses ephemeral keys. This protects the data from both active and passive attackers. Because the keys are single use, a single key disclosure does not compromise all past traffic. By using TLS, Tor also tries to hide itself from detection by emulating regular web traffic.

2.3 Circuits

Circuits are a series of nodes that a Tor client incrementally builds to create a virtual circuit of encrypted connections through the Tor network. As can be seen in Figure 2.2, the circuit consists of three hops. For the first hop, the Tor client first picks one of its three guard nodes, which are the main nodes the client uses to talk to the Tor network. These nodes stay the same for long periods of time for security reasons [36]. For the second hop, the Tor client picks a random node to use as its relay node and then extends the circuit through the guard node to the relay node. For the last hop, the Tor client picks a random exit node, a special node that allows traffic to leave the Tor network, and extends the circuit through the guard and relay node to the exit node. Since each node in the circuit is only aware of the previous hop and the next hop[1], no individual relay is aware of the complete path the circuit consists of. Once the circuit is set up, it can be used to transport streams.

Under the assumption that there is no global passive observer that can see *all* traffic and the nodes that the client has selected do not work together, it is not possible for a malicious node to link the client with the contents and destination of the traffic.

2.4 Streams

Streams are direct TCP-like connections over a circuit between the Tor client and the target server, usually a web server. Just like TCP, it is an ordered and reliable stream of bytes. The data in a stream is encrypted and decrypted while it is relayed along the circuit, with each node stripping off one additional layer of encryption. Once it reaches the exit node, the stream data is fully decrypted[2] and forwarded to the target server.

2.5 Cells

Cells are the messages that nodes use to communicate between each other. Whereas connections deal with streams of bytes (like in TCP), channels and circuits deal

[1] The guard node is only aware of the client and relay node. The relay node is only aware of the guard and exit node, and the exit node is only aware of the relay node and destination of the traffic.

[2] Although many applications, like web browsers, nowadays use their own encryption which would prevent the exit node from inspecting the data in the clear.

with cells, which can be compared to the datagrams in UDP, although cells have a fixed, predefined length. However, because they are sent over a TCP connection, they do have an ordering and delivery guarantee.

Cells come in two types: control cells and relay cells. Both cell types have a `circuit_id` field to associate them with a specific circuit. Control cells include a command that is to be interpreted by the node that receives them, for example a `connect` cell that is used to establish a circuit between two nodes. Control cells are encrypted with (only) the public key of the receiving node and thus only the receiving node is able to read the contents of the cell.

The other type of cells are relay cells. These cells have their command set to `relay` and contain additional headers before the payload, such as a `stream_id`, `digest` and `relay_cmd`. Whenever a relay cell is received, the cell is decrypted and the hash in the `digest` header is checked. If this hash matches the cell[3], this means the contents are successfully decrypted and the cell is interpreted by the receiving node. If the `digest` head does not match, there is still another layer of encryption around the relay cell. The receiving node looks up the next node using the `circuit_id` and relays the cell. This design guarantees that a Tor client can communicate securely with any node in the circuit while each node only knows the its predecessor and successor, but never the whole circuit. Because cells are padded to have a fixed length, it is not possible to infer what kind of command or content the cell contains by its length. A schematic view of the structure of a cell can be found in Figure 2.3.

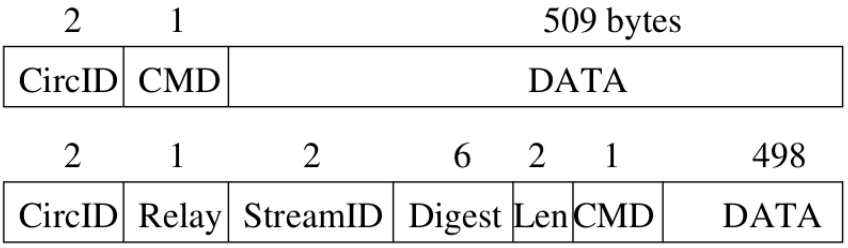


Figure 2.3: A schematic view of the structure of a Tor cell. At the top is a control cell: the first two bytes contain a circuit identifier (CircID) and the third byte is a one byte command (CMD). The remaining 509 bytes contain the payload (DATA). At the bottom is the relay cell. This is a cell with the third byte set to the `relay` command. Relay cells contain an additional header in front of their payload which consist of a two byte stream identifier (StreamID), a 6 byte digest over the payload (Digest), a 2 byte payload length field (Len) and a one byte relay command (CMD) field. The remaining 498 bytes contain the payload (DATA). Figure taken from [9].

[3]Since the `digest` field contains 48 bits, the chance of an accidental match is 2^{-48}, which is a practical impossibility.

Chapter 3

Internet protocols

To understand the underlying problems with Tor, a basic understanding about the used network protocols is required. The standards that are for network communications are divided in layers according to the Open Systems Interconnection (OSI) model. Figure Figure 3.1 shows a schematic view of the model and its layers.

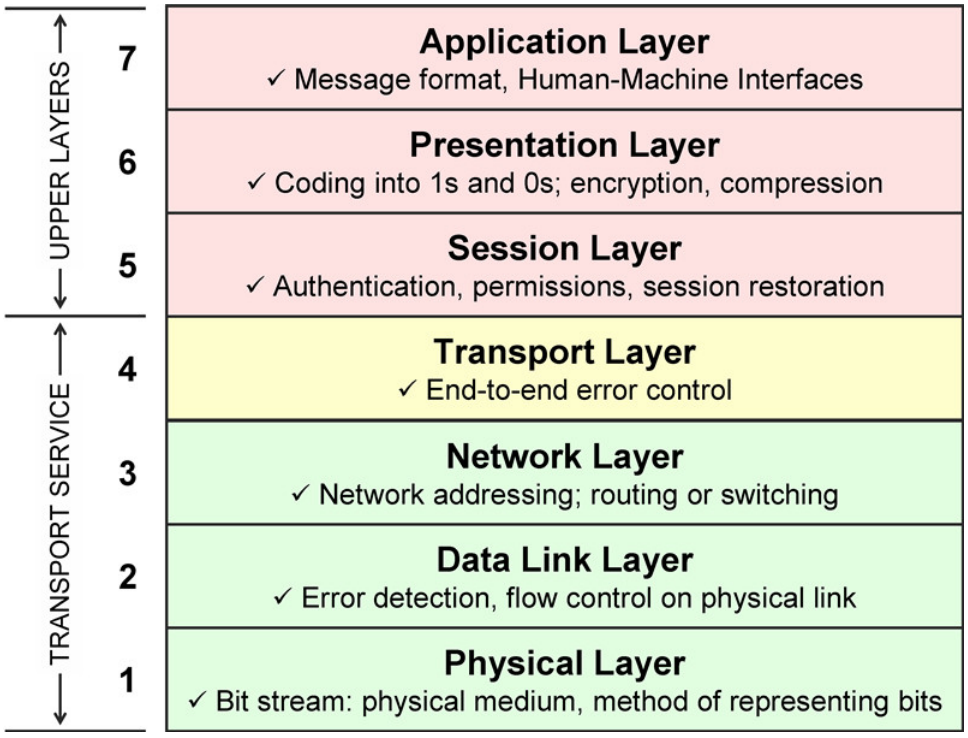


Figure 3.1: The Open Systems Interconnection (OSI) model that defines the abstraction layers that are used in network communications. Image taken from [7].

The two bottom layers contain the protocols to exchange simple data frames on a local network connection, such as Wi-Fi. On top of that is the network layer, which

is the layer that does IP addressing and routing. Tor uses IP addresses to address other nodes in the network. On top of that is the transport layer, which contains protocols to exchanges sequences of bytes between hosts. In the case of Tor, this is the TCP protocol that is used to set up connections between nodes. It corresponds to the connections as described in Section 2.1. The three upper layers in the model are usually handled by the application and in practice, the strict distinction between the three is not always well defined. In case of Tor, these layers contain the TLS session and the Tor protocol itself.

In the following sections, the main focus is on the protocols that are contained in layer 4, the transport layer, with in particular the TCP and UDP protocols in Sections 3.1 and 3.2. The head-of-line blocking and TCP congestion control problem that were identified in Section 1.1 are further explained in Sections 3.3 and 3.4. And finally, the proposed replacement transport layer protocol QUIC is detailed in Section 3.5.

3.1 Transmission Control Protocol (TCP)

TCP, or the Transmission Control Protocol [8], is one of the main Internet protocols. It is designed to send an ordered and reliable stream of data over the network between two endpoints. It provides limited data integrity using checksumming and uses ports for source and destination addressing. TCP is based on connections, and before two endpoints can communicate, a connection has to be setup between specifying a specific port on the other endpoint using a 3-way handshake. Data streams that are sent over the TCP network are put into packets with a header specifying several options of the packet and reassembled at the other endpoint. When a packet is dropped for whatever reason, TCP will detect this and transparently retransmit the dropped packets. This causes all other pending packets to be blocked until every preceding packet has been acknowledged. It also uses congestion and flow control to make optimal use of the network connection without oversaturating it.

Other than that, TCP presents itself as an interface to a ‘dumb’ data pipe that only sees a stream of bytes. Encryption and other features have to be implemented on the application level. If multiple independent connections are multiplexed over a single TCP connection, such as when tunnelling other TCP connections or requesting multiple files or pages over a single HTTP/2 connection, these independent connections can interfere with each other as described in Sections 3.3 and 3.4.

3.2 User Datagram Protocol (UDP)

UDP, or the User Datagram Protocol [45], mainly differs from TCP in the sense that it is not stream based. Instead it is a message based protocol that communicates using datagrams. Like TCP, it provides limited data integrity using checksumming and uses ports for source and destination addressing, but lacks all other features

that TCP provides. As a connectionless protocol, it has no concept of connections, and there is no handshake to setup a connection. It gives no guarantees in terms of delivery, ordering or duplicate detection, and leaves it up to the application to implement these features, if necessary.

Since UDP is little more than a simple header around datagrams, it is often used as a transport protocol to tunnel other existing protocols that are not widely supported over the Internet, or to implement new protocols without requiring support from intermediate routers, such as QUIC (see Section 3.5).

3.3 Head-of-line blocking

Reliable and in order delivered connections, such as TCP, work on a first-in first-out (FIFO) basis to preserve the order of the packets. A single dropped packet will block all other packets behind it until it is retransmitted. Every packet that arrives has to be acknowledged by the receiving party. TCP uses cumulative acknowledgement for this, which means that an acknowledgement for a single packet implicitly means that all packets that were transmitted before the acknowledged packet are also acknowledged. This reduces the number of acknowledgements, but also means that every packet that comes after a dropped packet will also be retransmitted, even if it has already been received.

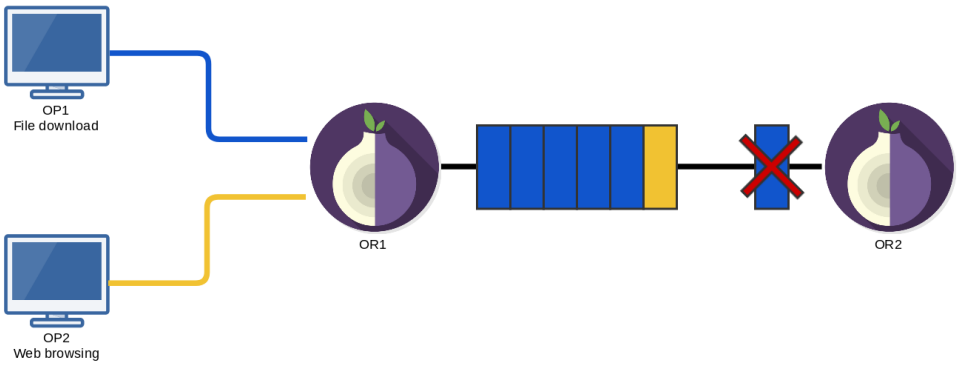


Figure 3.2: A schematic view of the head-of-line blocking problem. Clients OP_1 and OP_2 both have a circuit that goes from Tor relay OR_1 to OR_2. The rectangular blocks represent packets that are currently in transit, their colour corresponding to the originating client. OP_1 has a high latency, high bandwidth load, while OP_2 has a low latency, low bandwidth load. Now that one of OP_1 packets is lost, all packets already in transit are delayed until the lost packet is retransmitted. The packet from OP_2, unrelated with the dropped packet, is punished with additional, unnecessary delay.

The Head-of-line blocking problem applies when packets that are unrelated share the same FIFO buffer, in this case a TCP connection. An example is shown in Figure 3.2. In a worst case scenario, client OP_1 has a high bandwidth load

without a low latency network load, such as downloading a big file. This causes many packets to be transmitted, but latency is not an issue. Client OP_2 has a low bandwidth, low latency network load such as browsing webpages. As mentioned in Section 1.1, even a small additional latency is perceived as slow by many users.

Since Tor multiplexes the data of different circuits over a single TCP connection, they share the same FIFO buffer. The packets between node OR_1 and OR_2 show the packets that are in transit. Now that the first packet, originating from OP_1, is dropped, all packets that are already in transit will not be accepted by OR_2, even if the packet behind that originated from OP_2. Instead, OR_1 will first have to detect that the packet was lost (by lack of acknowledgement within a specific timeout) and then start retransmitting again from the lost packet. This means all packets after it, that were already in transit, will be retransmitted too. Only after successful retransmission of the packet for OP_1 will the packet for OP_2 be received by OR_2 with additional latency, even though those packets are independent of each other.

3.4 TCP congestion protocol

When a network connection gets oversaturated with more incoming traffic than it can handle, congestive collapse might occur. In this state, network performance is severely degraded by a large amount of dropped packets. To prevent this, TCP utilizes congestion control algorithms to slow down the transmission of packets when a network becomes saturated (or speed up again when possible). To do this it maintains a congestion window, which is the number of unacknowledged packets. Once the window is full, it stops transmitting packets until previously transmitted packets are acknowledged and free up the congestion window.

3.4.1 Congestion control and multiplexing

When a network connection becomes oversaturated, packets will start to drop. Most congestion control algorithms will take this as a signal to reduce the congestion control window. An example is shown in Figure 3.3. Here client OP_1 has a high bandwidth load without a low latency network load, such as downloading a big file. This causes many packets to be transmitted, but latency is not a large issue. Client OP_2 has a low bandwidth, low latency network load such as browsing webpages, for which increased latency is highly perceivable.

Since Tor multiplexes the data of different circuits over a single TCP connection, they share the same congestion window. Now when the network link between OR_1 and OR_2 starts to get saturated and packets start to get dropped, OR_1 will decrease its congestion window by halving it. Decreasing the congestion window leads to a lower bandwidth and a higher latency for both clients, even though it was the many packets of OP_1 that caused the reduction in window size.

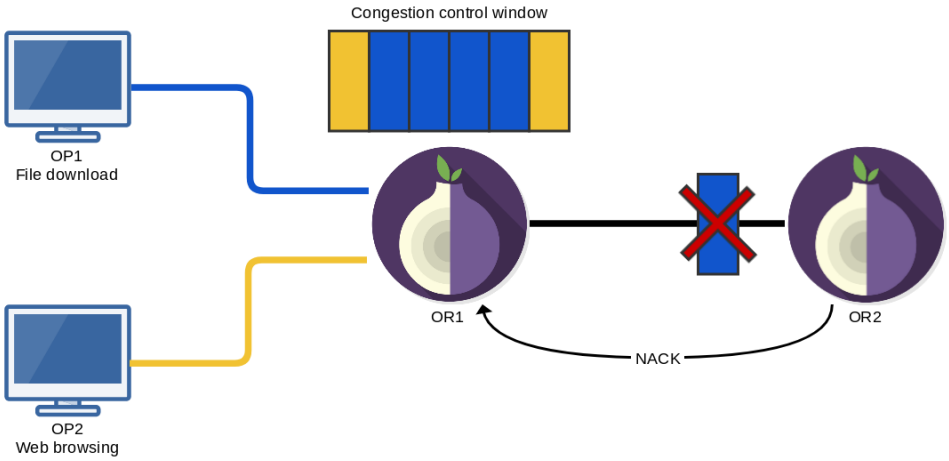


Figure 3.3: Schematic view of the TCP congestion control problem with multiplexing. Clients OP_1 and OP_2 both have a circuit that goes from Tor relay OR_1 to OR_2. The rectangular blocks represent packets, their colour corresponding to the originating client. OP_1 has a high latency, high bandwidth load, while OP_2 has a low latency, low bandwidth load. Now when the network link between OR_1 and OR_2 starts to get saturated and the shown packet is dropped, OR_1 will halve its congestion control window, leading to a lower bandwidth for both clients. OP_2 gets punished with reduced bandwidth because of the many unrelated packets from OR_1.

3.5 The QUIC transport protocol

QUIC is a protocol designed by Google in order to improve the latency of HTTP/2's multiplexed connections [50, 25]. After showing the improvements by deploying it in their own browser, it was adopted by the IETF under the IETF QUIC Working Group, which split the protocol into multiple parts. The QUIC transport layer and the HTTP-over-QUIC, also known as HTTP/3, are the two main parts [18].

The QUIC transport layer aims to be a replacement for the TCP protocol, providing (multiple) ordered and reliably streams of data. It is mainly focussed on reducing latency by reducing the number of round trips and by having native support for multiplexing multiple data streams into a single connection. It also has native support for encryption, based on TLS 1.3. Instead of relying on ports and IP addresses to keep track of connections, like TCP does, every connection has a unique identifier which is included in every packet. This allows endpoints to change IP address without having to reconnect. It is a UDP-based protocol and implements all its features on top of it.

3.5.1 Reducing round trips

TCP has a 3-way handshake and on top of that TLS needs an additional 3 or 4-way handshake[1]. In comparison, QUIC combines them into a single handshake by including the data to set up an encrypted session in the first packet. In order to do this, it needs to know some things about the other endpoint, such as the supported cipher suites. Since this is largely static data that does not change very often, it only has to be requested once and can be cached for every subsequent connection. The other endpoint can then respond with a certificate and, assuming the initiating endpoint accepts the certificate as valid, it can start sending encrypted data after a single round trip. A schematic illustration of the handshake can be found in Figure 3.4.

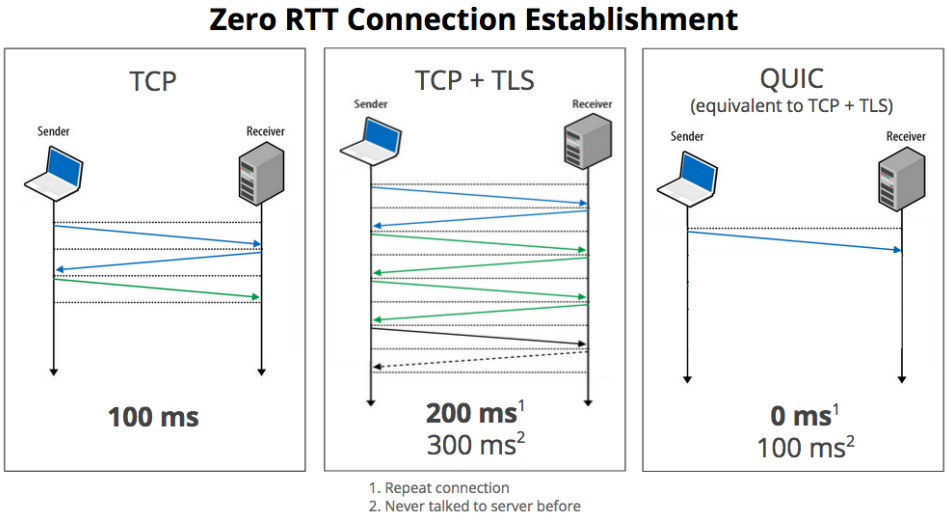


Figure 3.4: A schematic comparison of the handshake process between TCP, TCP+TLS and QUIC. Most secure connections, such as HTTPS web traffic and Tor connections between nodes, use the TCP+TLS handshake to set up a connection. Because the connection and encryption layer are strictly separated, the TLS handshake can only be started once the TCP connection handshake is fully completed. QUIC combines the two layers and, in the best case, is able to immediately start sending data with the first packet, greatly reducing the time needed to set up a secure connection. Image taken from [4].

In order to reduce the number of round trips for subsequent new connections even more, the client can also use a special cookie provided previously by the server. Because the client already has the certificate of the server, it can skip the first packet and immediately start sending encrypted data without a single round trip. It includes this cookie in the packet in order to authenticate itself to the server. This is similar to the 0-RTT handshake in TLS 1.3. However, some concerns have

[1] Depending on the version and features like 0-RTT.

been raised about the security of this method, for example because of the lack of perfect forward secrecy or the protection against replay attacks [13].

3.5.2 Reducing retransmissions and blocking

Because QUIC is a UDP-based protocol which has no loss detection or retransmission support, it implements those as part of the protocol. Because QUIC has native support for multiplexing multiple streams of bytes over a single connection, it also implements retransmission on a per-stream basis. If a single packet is dropped, only the stream that the packet belongs to has to wait for retransmission. This solves the head-of-line blocking issue of TCP detailed in Section 3.3, without the loss of performance or security. A schematic view of the differences can be found in Figure 3.5.

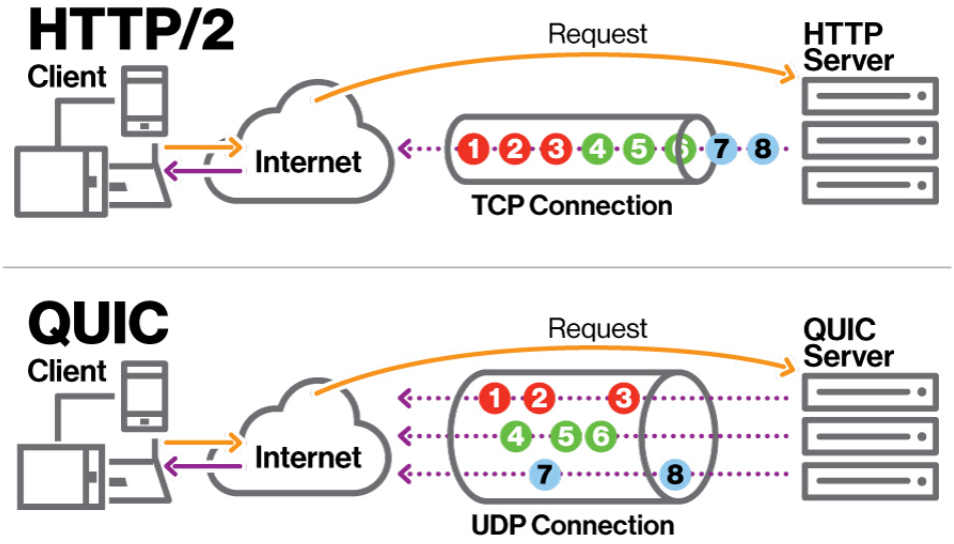


Figure 3.5: At the top is a normal HTTP/2 client multiplexing three different requests over a single TCP connection. Because TCP only supports one single stream at a time, the requests are handled sequentially. At the bottom, a QUIC client multiplexing the same three requests over a single UDP connection. Because UDP is unordered and QUIC supports multiple streams at the same time, requests are handled in parallel and do not interfere with each other. Image taken from [44].

Congestion control is also implemented on a per-stream basis and could even use different algorithms for different network loads. This solves the TCP congestion control issue as detailed in Section 3.4. An additional way that is being explored to reduce retransmission is to make use of the unused space in packets and include Forward Error Correction information in that space. This would make it possible to reconstruct a dropped packet with the data from other packets, preventing the need to retransmit that packet entirely. Another improvement over TCP and TLS is

to encrypt all packets individually. This is not possible in TCP because it exposes a stream of bytes to the TLS layer. The TLS layer has no knowledge about where a packet starts or ends. Because QUIC encrypts all packets encrypts individually, it is possible to decrypt them while previous packets are still being retransmitted.

Chapter 4

Related work

Others have tried to solve the problems performance problems in Tor in various ways, ranging from relatively small transport layer changes to almost complete overhauls of the network model. To give an overview of the previous research, some of them are detailed in the sections below.

4.1 IPSec Tor

Kiraly et al. [30] showed the performance penalties of running TCP over TCP tunnels and propose using a Layer 3 approach with a combination of NAPT and IPSec to solve these performance issues. Instead of using TLS over TCP, the connections between nodes are encrypted on a lower level, by using the kernel space implementation of IPSec. When used in tunnel mode, this encrypts the full IP packet, including the header, making the destination IP and port of the packet confidential. Instead of using circuit ids like in the current implementation of Tor, they propose using the NAPT (`dest_ip`, `trans_proto`, `dest_port`) tuple as a label for which circuit the packets belong to. This has the advantage that you have a real network that can transport TCP, UDP and any other transport layer protocol.

However, this is also a disadvantage. Because the packets in the transport layer are forwarded as-is, it is now possible to fingerprint the originating network stack at the destination side of the circuit. Tools are readily available for this purpose, such as the OS detection feature in the `nmap` utility [51]. In addition to that, while relying on the host Operating Systems IPSec implementation for encryption might give a slight performance boost because there is less time spent on moving bytes between user space and kernel space, it could also reduce the security and portability. Not all hosts might have the latest updates applied and the kernel has to be trusted to correctly apply all encryption which might reduce security and because not all Operating Systems might have a compatible IPSec implementation available it might reduce portability. In addition to that, the deployability of this solution is very low because it would require setting up a whole new network, without any backwards compatibility with the existing Tor network.

4.2 DTLS Tor

Joel [49] noted the same problems with Tor as described in Chapter 3 and proposed to use TCP-over-DTLS to improve the performance. This is done by replacing all system TCP sockets with user space TCP sockets and wrapping the resulting TCP data inside a DTLS packet, a protocol which uses the TLS protocol over UDP datagrams. Since the packets are now wrapped inside a DTLS packet, it is possible to establish a new TCP connection for each circuit instead of multiplexing them over a single TCP connection without the security and performance penalties as stated in Section 2.1. This solution is quite similar to how a QUIC-based implementation of Tor would be, although it does not solve the latency penalties such as the ones that are associated multiple round trips to set up a secure connection. In fact, the QUIC design document [50] states that:

> The eventual protocol may likely strongly resemble SCTP, using encryption strongly resembling DTLS, running atop UDP.

In that quote SCTP is used to multiplex different streams over a single connection instead of multiple TCP connections, but the idea is similar. It seems that using QUIC is an incremental improvement over the TCP-over-DTLS design.

4.3 uTor

Nowlan et al. [41] noted that the strict in-order delivery of TCP was one of the contributing factors of why Tor is slow and proposed using unordered TLS (uTLS) over unordered TCP (uTCP) to improve the performance. Limited evaluation seems to suggest that this indeed solves the head-of-line-blocking problem as described in Section 3.3. While the solution has the advantage that this only requires a very limited amount of code to be changed in the Tor code base and it does not alter wire-format of the traffic an outside observer would observe (in other words, it would not interfere with the ability of Tor to blend in with normal TLS traffic), it has the major downside that using uTLS requires a kernel patch which makes the probability of it ever being widely deployed very unlikely, even more so if this is to be supported on the client side as well.

4.4 QuicTor

Ku et al. [32] also noted the current performance issues of Tor as described in Chapter 3, and also found that an UDP-based network protocol might be a way to solve those performance issues and found that QUIC might be a good candidate. At the time, the QUIC protocol was still a Google-only experiment that they were testing in their own web browser and was not yet adopted by the IETF working group in order to be standardized. They made a prototype by wrapping the Chromium

QUIC network stack in a custom and now unmaintained wrapper to simulate a TCP socket-like API and integrated it into a now obsolete version of Tor.

Additionally they had a flaw in their implementation: the identifier that Tor gives to the end-to-end streams between a Tor client and an exit node (via the circuit) were reused as the `stream_id` values of the QUIC streams. This has the benefit that streams using the same circuit do not interfere with each other, but this has two drawbacks: first of all, the Tor stream identifiers are only known to the beginning and end of the Tor stream, not the nodes in between. The relaying nodes have the Tor stream identifier default to zero, meaning that *all* the node traffic would then share the same QUIC stream number zero. Secondly, if all Tor streams would use their own QUIC stream, this would leak information to the relaying nodes as they can now distinguish between different streams inside a circuit, increasing the chance of linkability.

Despite that and the limited performance and security analyses, the results still suggest that using QUIC as the UDP-based transport protocol in Tor could indeed an improvement over the current TCP-based Tor implementation and a topic worth researching.

Chapter 5

Integrating QUIC in Tor

There are multiple ways to switch out the current TCP-based transport layer protocol that Tor uses. There are multiple options to as to what to replace TCP with, as only UDP does not provide many of the features that Tor relies on, such as managing connections and congestion control. Additionally, there is a choice between implementing these features hop-to-hop or end-to-end. In the former case, the connection and channel layers of the Tor network model (see Chapter 2) are modified and only the direct connections between nodes use the new UDP-based protocol. The second option would mean implementing them on the stream layer of the Tor network model, simulating a real end-to-end connection, and doing no (or less) ordering, congestion control and other network features on the lower levels.

In the first part of this chapter, a list of design requirements is formulated that the new design will have to comply with. Based upon these requirements, a case is made why QUIC was chosen as the new transport layer protocol and the decision to use hop-to-hop instead of end-to-end is explained. Finally, it is described how these design choices will fit in the current Tor network architecture.

5.1 Design requirements

In order for the design to be a clear improvement over the current version of the Tor implementation, several requirements have been formulated. They are based on the observations made about the shortcomings in the previous chapters and .

5.1.1 Head-of-line blocking

In the current implementation of Tor, several circuits are multiplexed over a single TCP connection. One of the observations of Dingledine et al. was that this is a contributing factor as to why Tor is slow [10].

Independent Tor cells that belong to different circuits and that are multiplexed over the same connection should not block unrelated cells from being delivered,

for example when a cell is dropped. In other words, the design should solve the problem called head-of-line blocking, as detailed in Section 3.3.

5.1.2 TCP congestion control

As described in the previous requirement, independent Tor cells that belong to different circuits that are multiplexed over the same TCP connection also share the same TCP congestion control algorithm and congestion window. Whenever it resizes the congestion window, it applies to all circuits.

Independent Tor cells that belong to different circuits and that are multiplexed over the same connection should not share the same congestion window and should not slow down other circuits when they oversaturate the network link. In other words, the design should solve the TCP congestion control problem as detailed in Section 3.4.

5.1.3 Performance

Since the goal of the design is to improve the performance of Tor, one of the key requirements is the ability to evaluate the performance and to verify it is actually improved. The performance of normal use of the Tor network should not be negatively impacted.

There are two metrics on which the performance will be evaluated. First is the *time to first byte*, which is the time it takes between initiating a connection and getting the *first* byte from the remote server, which takes a full round trip. The second metric is the *time to last byte*, which is the time between initiating a connection and getting the *last* byte from the remote server, after which the connection is usually closed. For the exact definitions used during the evaluation, see the descriptions in Section 8.1.

5.1.4 Privacy and anonymity

As the anonymity of its users is one of the core functions of the Tor network, the proposed design should not introduce any problems or features that increases the chance of linkability or decrease the anonymity of the users of the Tor network in any other way. For example, just not multiplexing different circuits over the same TCP connection between nodes and instead opening a new TCP connection for each circuit could improve the two main performance problems that were identified earlier. However, it is not a valid solution, one of the reasons being that this could increase the linkability of the circuits because they are now distinguishable to a passive observer.

5.1.5 Deployability

The solution should be deployable in stages. As the Tor is a decentralized distributed system, different versions of the software run side by side and not every

node in the network will or is able to update to the latest version in a timely matter. While a significant part of the network uses a reasonably recent version of Tor, over one sixth of the current Tor relays still use a version that is over 3 years old, while the oldest version that is still being reported was first released six years ago [48].

In other words, the design should be backwards compatible with existing nodes. If nodes that support QUIC are incompatible with older versions of Tor there will be a network split which will severely decrease the chance of it ever being deployed.

5.2 Why QUIC?

HTTP connections, which are used by web browsers, have a similar problem as the Tor network has: websites consist of different elements that are requested separately, often with a wide variety in size. Loading the text of a web page has high priority but is usually only a small payload, while downloading big media files usually has a lower priority but is a big payload. While historically browsers would open multiple connections and then sequentially send requests over these connections, waiting for the request to complete before sending a new one, the more modern HTTP/2 standard multiplexes these requests into a single connection. But, as HTTP/2 is still using TCP, it suffers from the same drawbacks as Tor when multiplexing different circuits over a single TCP connection. Work has been done by Google to create a new UDP-based transport protocol for the next version of HTTP/3 under the name of QUIC [50].

As described in Section 3.5, the QUIC protocol natively supports multiplexing independent data streams over a single connection. This allows it to avoid the head-of-line problem as data that is associated with different streams can be received out of order. It also has support for congestion control which is independent for each of the multiplexed streams, which allows it to avoid the TCP congestion control problem. Additionally, it also uses (modern) TLS to encrypt the connection, just like Tor uses as its outer layer encryption. Because TLS is integrated directly into the protocol, it can also reduce latency by doing the connection and secure session handshake at the same time, instead of sequentially like when setting up a secure TLS session over TCP. Since it works over UDP instead of TCP, it is possible to listen on both ports to maintain backwards compatibility with Tor nodes that do not support QUIC yet.

All this together means that QUIC can replace multiple parts of the Tor network layers as described in Chapter 2.

5.3 Hop-to-hop

Instead of relaying the QUIC connection over the circuits and making it end-to-end, the decision was made to only establish the QUIC connection hop-to-hop. This is because it has multiple advantages over the end-to-end approach, while still

solving the two main performance problems that were identified in the old design. First of all, hop-to-hop has the advantage of being less of a change to the current Tor network model than making the QUIC connection end-to-end. This makes the change less invasive with less chance of introducing new unintended security vulnerabilities into the design of the network, as well as reducing the implementation complexity.

Making the QUIC connection dependent on only two communicating nodes instead of all the nodes in a circuit also greatly improves the deployability of the design. In an end-to-end design, all intermediate nodes in a circuit will have to be upgraded to a QUIC-capable version or else the whole circuit will have to fall back on the old design. As noted while defining the deployability requirement in Section 5.1.5, there is still a significant part of public nodes in the Tor network that runs an older version. There is a trade-off to be made here by the Tor client: does it favour security over performance by building a circuit from all possible nodes, likely falling back to the older, legacy circuits, or does it favour performance over security by building a circuit from only QUIC-capable nodes, reducing the number of nodes it can pick from?

But even when most or even all public nodes in the network are updated to a newer, QUIC-capable version and the client does not have to pick from a reduced set of nodes to get performance improvements, an end-to-end design is also dependent of the number of clients that are capable of using the new QUIC-based circuits. Because not only does an end-to-end design expose additional information about a client to all intermediate nodes in a circuit[1], it also exposes extra information to a passive observer which can observe the traffic between certain nodes, because it can now differentiate between newer QUIC-capable and older non-QUIC-capable clients. This could especially increase linkability if your client is either part of a small number of clients that already builds the new end-to-end QUIC-based circuits when not many clients have updated, or is part of a small number of clients that still builds the old legacy circuits when most clients have updated. In an end-to-end design, this problem can only be fully avoided by switching the whole network from one type to the other all at once. But again considering that a significant part of the network still runs a Tor version which was first released many years ago, this could either take many, many years or this would mean locking out part of the network until they upgrade.

In contrast, in the case of a hop-to-hop design, any connection between nodes will opportunistically switch over to a QUIC-based transport protocol when both sides have upgraded their Tor version. Even clients that have not updated yet will benefit from the performance improvements when they build a circuit with QUIC-capable nodes, although the connection from the client to the guard node might still benefit from a QUIC-based transport. While this will likely improve in the fu-

[1] Because now the relay and exit node have an additional bit of information about the, unknown to them, initiator of the circuit: is it running a Tor version new enough to use the new QUIC-based circuits or not?

ture as HTTP/3 will be introduced and becomes popular, some restricted networks might block UDP traffic while still allowing TCP traffic that looks like normal web browsing. A client on such a censored or restricted network will have to fall back on the old implementation without any of the performance benefits in and end-to-end design, while in a hop-to-hop design it can still benefit.

5.4 Tor network layers

Figure 5.1 is an updated version of Figure 2.2, showing the new layers that are used when using QUIC. First, the TCP connection between nodes is replaced by an UDP connection. On top of the connection, there is still an encrypted TLS layer, although in the new case this TLS layer is now part of the QUIC connection and therefore handled by the QUIC library instead of as a separate layer. This is illustrated by having a QUIC channel instead of a TLS channel.

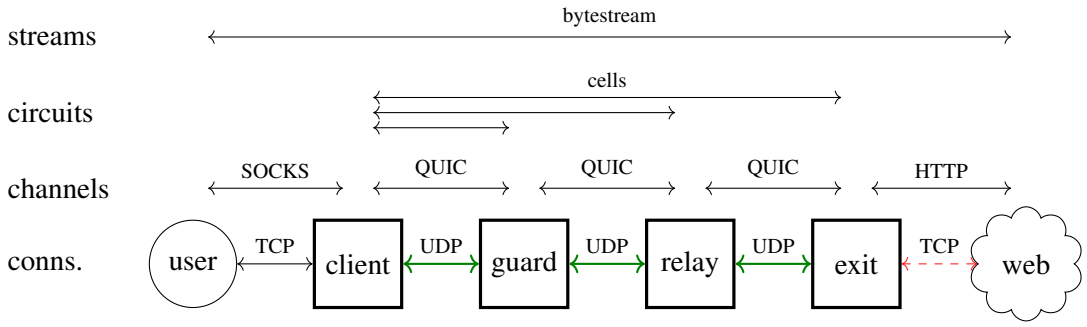


Figure 5.1: An updated version of Figure 2.2, showing a schematic view of the Tor network layers when using QUIC. In this updated figure, the TCP connections between the Tor nodes are replaced by UDP connections. On top of the connections is still a layer of TLS encryption, but this is now part of the QUIC protocol. This is illustrated by replacing the TLS channels with a new QUIC channel. Their behaviour is the same to the upper layers, other than that the `circuit_id` is reused to multiplex independent streams for each circuit.

In the original Tor network layers, channels pass the cells they send between nodes in order, due to the nature of the underlying TCP connection. The order of the cells can not change during transit[2] and the channels themselves do not care about the contents of the cells. In the new situation, channels will read the `circuit_id` header of the cell and use the `circuit_id` to create independent QUIC streams for each circuit. In practice this means that each circuit gets its own independent byte stream inside the QUIC connection, without the performance overhead or security loss of multiple TCP connections as described in Section 2.1.

[2]The packets carrying the cells can, but the TCP stack puts them back into order, making cells wait for their predecessors.

Chapter 6

Implementing QUIC in Tor

Based on the requirements and the design decisions in Chapter 5, a prototype was implemented to evaluate the performance improvements of using a QUIC-based transport protocol instead of a TCP-based transport protocol. One of the main implementation choices is which QUIC library to use, as this has a big impact on the implementation. The requirements and the subsequent library choice are detailed in Section 6.1. A description of how the library and QUIC protocol fit in the current Tor implementation is described in Section 6.2.

6.1 QUIC libraries

Due to the as of yet still experimental status of the QUIC specification, there are no mature or go-to implementations yet, nor a standard API that libraries can implement. The libraries that do exist are of varying maturity and interoperability between different libraries is not guaranteed and often lacking. There is no guarantee that clients and servers can communicate unless they use the same version of the same library. There are currently 22 different implementations listed on the IETF QUIC Working Group wiki that implement the "IETF QUIC transport" standard [19].

The requirements for the QUIC library are:

- Integration: The library should be accessible from Tor as a library via a standard interface. In practice, this means that it should expose a C compatible interface.
- Platform: The library should at least run on Linux, as this is by far the most used platform for public Tor relay nodes [48]. Ideally, it should also run on popular client platforms such as Windows and macOS.
- License: The library should be open source under a license which is compatible with the Tor license and should allow Tor to link to it and distribute it (for example with the Tor Browser bundle).

- Dependencies: The library should not have other unusual, unmaintained, or hard-to-use dependencies.

- Maturity: The library should actually work and implement the important parts of the specification, and be able to both act as server and as client.

- Maintained: The library should be actively maintained to lower the chance of bitrot and the need to invest into switching to a different library in the future.

6.1.1 Quiche

The Quiche library is an open source implementation of QUIC in Rust, a compiled systems language focused on safety, particularly safe concurrency and memory safety. Since the Rust language provides the tools create a C compatible interface. Additionally, since the Tor code base is slowly moving towards integrating more and more Rust code [47], this seems like a future-proof choice. The external interface of the library is relatively simple: It has functions to create and modify a connection configuration object and TCP-like functions to create and accept connection objects. The actual I/O, reading and writing to a socket, is left to the application. This makes it possible to reuse the existing socket functions and abstractions in Tor. For all these reasons, Quiche satisfies the integration requirement.

In addition to Linux, the library also builds on iOS, Android, macOS and Windows, making it likely that it supports most, if not all platforms that Tor supports, which makes it satisfy the platform requirement.

Quiche is licensed under the 2-Clause BSD License, which is a very permissive license whose only requirements are to include a copyright notice and disclaimer while either distributing the library source or as a binary. This satisfies all the license requirements.

It has two major dependencies: the Rust cryptography library *ring* and the TLS library *BoringSSL*. Ring is well maintained with regular updates, while BoringSSL is a OpenSSL fork by Google, that is used as their go-to TLS implementation (see Section 6.2.3). Since these are both widely used and well maintained, this satisfies the dependencies requirement.

The Quiche library itself is developed at Cloudflare, one of the biggest content delivery networks. It is also well maintained with regular commits and implementing the latest QUIC specifications. According to the libraries' `README.md` file, it is used in production at Cloudflare [6]. This makes it likely that this library will keep being maintained, satisfying the last two requirements.

In conclusion, seeing as the Quiche library satisfies all the previously defined requirements, it was chosen as the QUIC library.

6.2 Tor connection model

The other main implementation challenge is how to fit the design choices made in Chapter 5 into the current Tor implementation. The following subsections go into detail about the parts of Tor that were changed and how they fit into the new QUIC design.

6.2.1 Connections

Recall from Section 2.1 that the lowest layer of the Tor network model are connections. Tor implements these connections with an abstract `connection_t` object, whose interface is implemented by connection specific implementations depending on their use, such as the `or_connection_t` for Onion Router connections (connections between nodes) or the `edge_connection_t` for connections leaving the network, such as in exit nodes. All these connections use TCP as their underlying network protocol [1].

When Tor opens a port to accept connections, it creates a special TCP listener connection on a specific port. Libevent is used to monitor the opened socket for incoming data. Whenever another Tor node wants to open a connection on this port, libevent fires the read event on the listener connection. The functions that handles this read event then uses the `accept()` system call which creates a new TCP socket and wraps it in a new `connection_t` object. With the connection established, this is its own independent connection and any communication over this new socket is independent of the original listener connection and socket. Any time some new bytes arrive for this connection, a read event is raised and libevent calls the read function specific for this connection. A schematic view of this process is found in Figure 6.1.

As QUIC uses a UDP socket, this model does not apply because UDP lacks the concept of connections (see Section 3.2). Instead, any time the UDP socket receives a datagram, libevent fires the read function for the listener connection, which loads the datagram into a buffer and passes it to the QUIC library. The QUIC library then parses the header inside the buffer. In case the packet was internal and requires a response, such as packets for establishing or maintaining connections, the library queues up new QUIC packets in its outgoing buffer. If the packet contained actual data, the library signals that one of the connections is readable. The listener connection has to read the packets `connection_id`[2], and either has to create a new `connection_t` object (linked to the same listener socket) or find a matching existing one. For this purpose, the listener connection has to maintain a list of connection objects and their `connection_id`. It then

[1] Except the DNS port listener, but this is a separate implementation that only implements the DNS protocol.

[2] The `connection_id` is a replacement for the originating $(IP, port)$ tuple that TCP and UDP use. It is encrypted to prevent spoofing and makes it possible to keep the connection going when one of the hosts IP address changes.

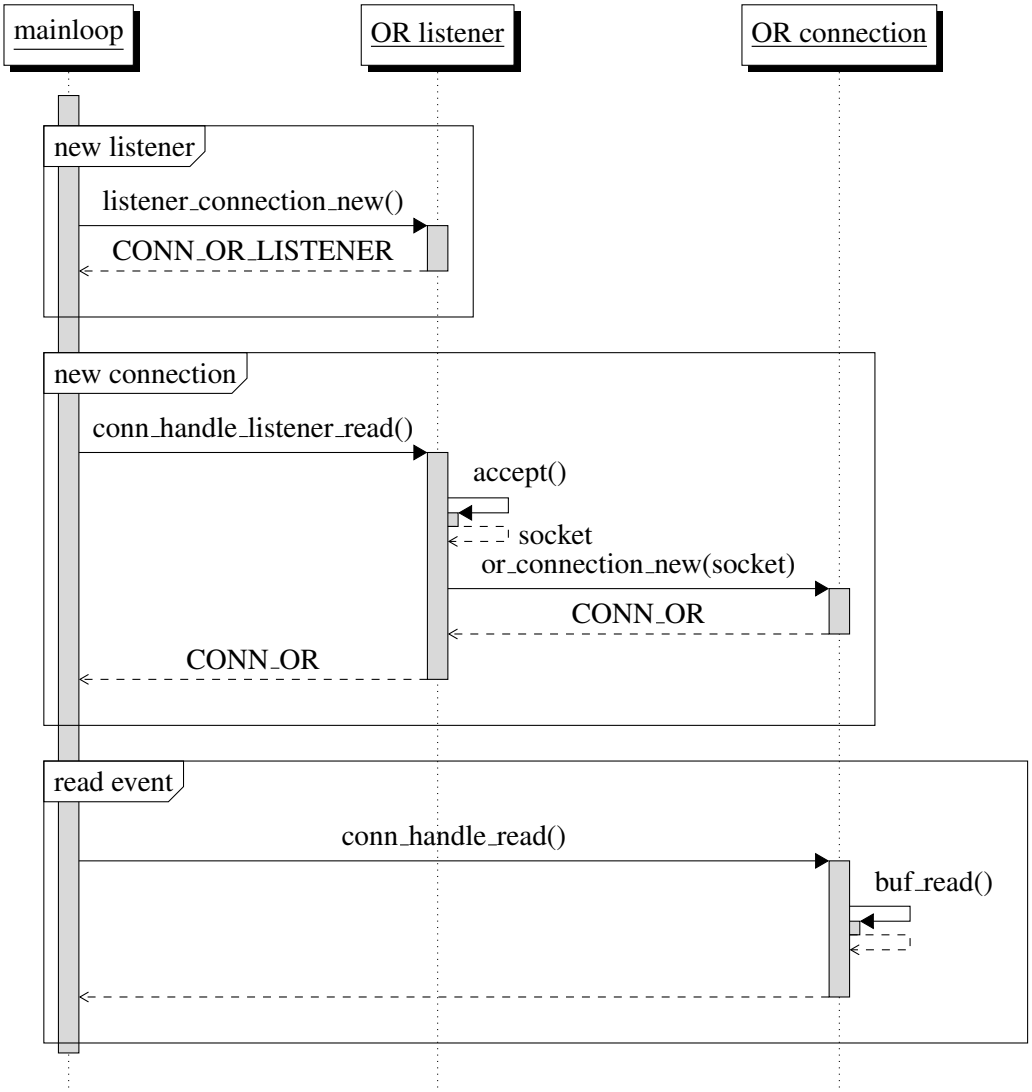


Figure 6.1: A schematic view of the sequence opening and reading TCP-based OR connections. In the first block, a new TCP socket is opened and a new `connection_t` OR listener is returned. In the second block, when a read event is fired on this socket, a new socket is created with the `accept()` syscall. This socket is used to create a new `connection_t` OR connection. The third block shows that a read event on this new socket is handled directly by the OR connection.

passes the connection to the read function to handle the incoming data and tries to write any waiting QUIC packages to the UDP socket. A schematic view of this process is found in Figure 6.2.

In contrast to the original listener connection, the QUIC listener needs to keep

its socket open until all other associated connections are closed, because they all share the same socket.

6.2.2 TLSChannel

As mentioned in Section 2.2, channels are one of the layers of the Tor protocol. They are an internal only layer that provides a cell-based interface over the raw connections between nodes. The current Tor implementation tries to implement an abstract channel interface in the `channel_t` object. Specialized channels can then implement this interface without having to modify all code dealing with channels. However, the only implementation is currently the `tlschannel_t` object and the abstraction between the `channel` and `tlschannel` is often violated[3]. More than once, TLS specific functions are called outside the `tlschannel` implementation, making it non-trivial to swap out the `tlschannel` implementation with something different.

Additionally, the Quiche also handles the TLS layer of the connection. This means that, instead of handling the TLS handshake and connection setup in the `tlschannel`, this is done automatically when a QUIC connection is established.

Because of these two reasons combined, the current prototype modifies the existing `tlschannel` object to bypass the TLS initialization and to call the QUIC related functions to send and receive buffers that contain cells, based on a flag set on the connection objects that determines if the connection uses QUIC or not.

6.2.3 TLS libraries

The current Tor implementation supports two different TLS libraries: the widely used OpenSSL library and Mozilla's Network Security Services (NSS) library. By default however, OpenSSL is used. These two libraries are abstracted into a common internal API defined in `tortls.c`, respectively in `tortls_openssl.c` and `tortls_nss.c`. QUIC uses TLS as encryption layer, but it uses a modified handshake for which the currently used TLS libraries have no API to support this yet. For this reason, the Quiche library that provides the used QUIC implementation uses an OpenSSL fork by Google called BoringSSL.

This is not a perfect or permanent solution however. To quote the BoringSSL manual [17]:

> Although BoringSSL is an open source project, it is not intended for general use, as OpenSSL is. We don't recommend that third parties depend upon it. Doing so is likely to be frustrating because there are no guarantees of API or ABI stability.

This is also a reason to avoid implementing a `tortls_boringssl.c` abstraction that would avoid using multiple TLS libraries at the same time and only use

[3] As noted in this, at the moment of writing, still open ticket: `https://trac.torproject.org/projects/tor/ticket/23993`.

BoringSSL for all TLS operations in Tor. For this reason, until OpenSSL gains support for the QUIC handshake, both libraries will have to be used side-by-side. However, since BoringSSL is a OpenSSL fork, they still share much of the same symbols. Linking both directly in Tor is not possible. To keep these separated, the Quiche library needs to be built as a shared library that only exposes the external QUIC connection API instead.

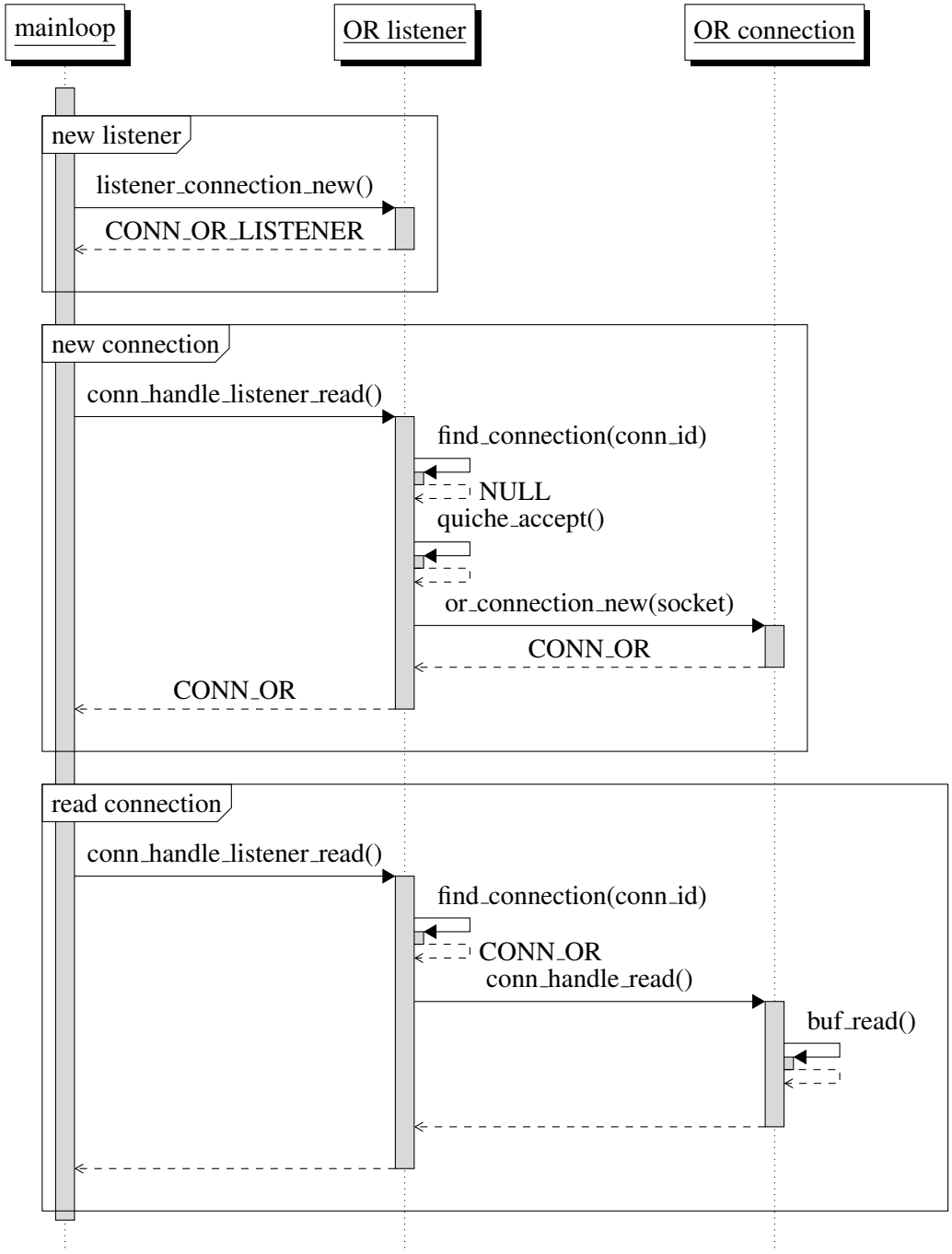


Figure 6.2: A schematic view of the sequence opening and reading QUIC-based OR connections. In the first block, a new UDP socket is opened and a new `connection_t` OR listener is returned. In the second block, when a read event is fired on this socket, the OR listener tries to find an existing connection associated with the incoming packet based on the `connection_id`. None were found, so a new `connection_t` OR connection is created, linked to the listener socket. The third block shows when a read event on the listener socket does find an existing connection, and calls the read event of the OR connection.

Chapter 7

Tor network simulation

To evaluate the QUIC-based implementation of Tor against the requirements as defined in Section 5.1 and compare it to the existing TCP-based implementation, a virtual Tor network will be simulated and tested. In the following subsections the available simulators that were used and considered are detailed.

7.1 Existing simulators

Over the years, multiple ways were developed to test modifications to the Tor network. While their implementation and features differ, they all have the goal to run the Tor client in a virtual network to observe its behaviour. As there is no need to reinvent the wheel, several of these options were first tested and considered.

7.1.1 Shadow

Shadow [28] is a network simulator that was built to test large scale Tor networks on a single machine in a local, isolated environment. Instead of using the hosts real in-kernel network stack it simulates a virtual network stack in user space.

To define the network that will be simulated, Shadow uses GraphML network topology files. These network topology files are XML files that define a directed or undirected graph with a list of nodes and edges. Nodes represent either a single client or a network cluster, while edges are the network links between nodes. Each node can have various properties set such as limited up and down bandwidth or a certain percentage of packet loss. When the node is a cluster, this bandwidth is shared with all the Tor clients that are linked to this cluster. When the node is a single client, it only applies to that specific client. The network links between nodes that are defined by the edges of the graph can also have a number of properties, such as network latency and packet loss. If there is a direct edge between two nodes, this edge will be used as a network link to communicate. In case there is no direct edge, Shadow will use Dijkstra's algorithm to find the shortest path between the two nodes. An example of a very minimal topology can be found in Figure 7.2.

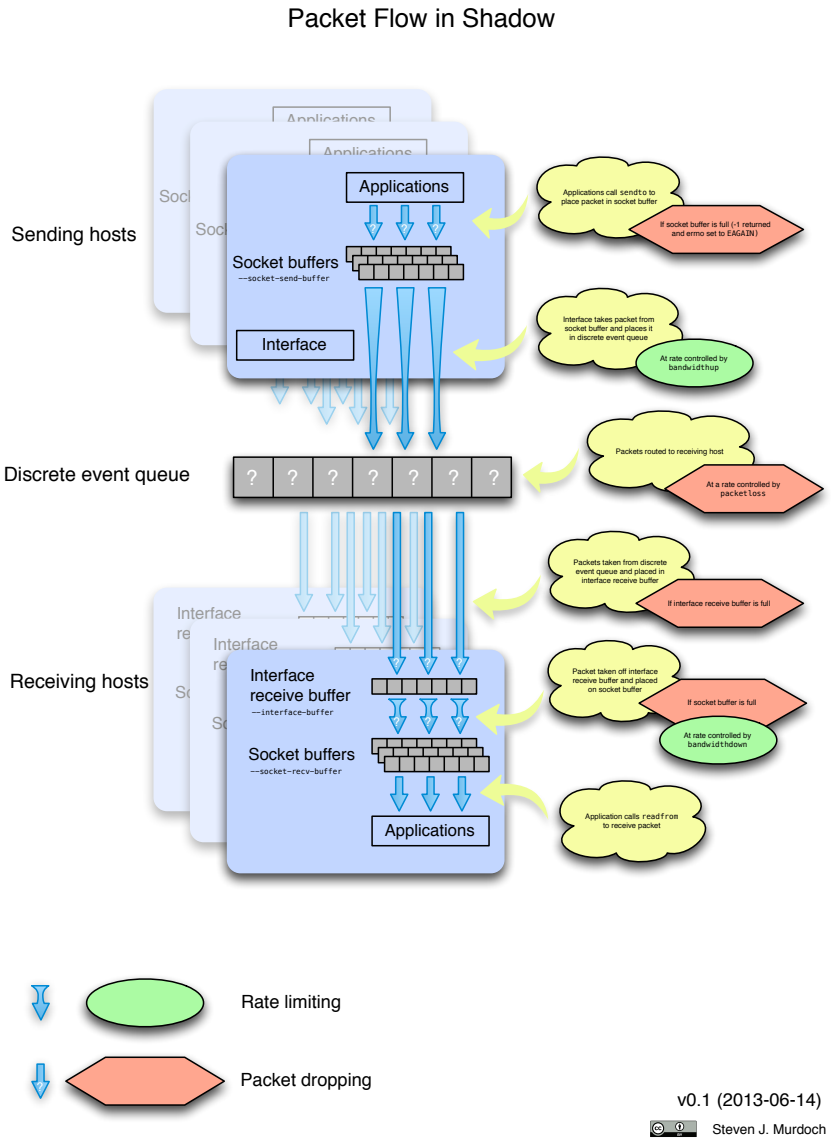


Figure 7.1: A schematic overview of how the packet flow in Shadow works. Applications are modified so that all network syscalls can be hooked and routed through a virtualized network instead. Packets flow from application specific socket buffers to a shared a discrete event queue, at which point some packet drop is applied and they are routed to first the receive buffer of the receiving application and finally the socket buffer, at which point the hooked syscalls will receive them. All this is done in user space, without the hosts network stack. Image by Steven Murdoch [26].

```
<?xml version="1.0" encoding="utf-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <key attr.name="packetloss" attr.type="double" for="edge" id="d6" />
  <key attr.name="latency" attr.type="double" for="edge" id="d5" />
  <key attr.name="packetloss" attr.type="double" for="node" id="d4" />
  <key attr.name="countrycode" attr.type="string" for="node" id="d3" />
  <key attr.name="bandwidthdown" attr.type="int" for="node" id="d2" />
  <key attr.name="bandwidthup" attr.type="int" for="node" id="d1" />
  <key attr.name="ip" attr.type="string" for="node" id="d0" />
  <graph edgedefault="undirected">
    <node id="poi-1">
      <data key="d0">0.0.0.0</data>
      <data key="d1">2251</data>
      <data key="d2">17038</data>
      <data key="d3">US</data>
      <data key="d4">0.0</data>
    </node>
    <edge source="poi-1" target="poi-1">
      <data key="d5">50.0</data>
      <data key="d6">0.05</data>
    </edge>
  </graph>
</graphml>
```

Figure 7.2: A very minimal GraphML network topology file as used by Shadow. It defines a single node `poi-1` with various properties such as a limited amount of up and down bandwidth and no packet loss. The edge connects the node to itself through a network connection with 50 ms latency and a 5% packet loss, as defined in key `d5` and `d6`.

While Shadow was originally developed with Tor in mind, it can be extended for use with other distributed and peer-to-peer systems through the use of plugins. Currently, only a Tor plugin and a Bitcoin plugin are available [26]. Although Shadow requires a plugin for an application to work with Shadow, it does not require any Shadow specific modifications to the source code of the application itself. However, in order to hook into the application, this means that Shadow does require access to the source code of the application and does not work with existing, unmodified and general purpose binaries.

Because Shadow does not use the host's real network stack, it instead hooks into the application to simulate the underlying network stack, which has definite performance advantages. It also has the advantage that the network behaves the same on every computer, there is no interference with the host network and there are no root permissions required to run Shadow. However, it also means that it only has support for the network protocols it explicitly supports. Currently, Shadow only supports TCP which means that testing a Tor network based on a UDP-based protocol such as QUIC is not possible with Shadow.

7.1.2 Chutney

Chutney [46] is a collection of scripts to make it easier to do local testing of the Tor network. In contrast to the other network simulators, it doesn't do actual simulation or emulation of the underlying network. All Tor clients run on the same machine over the loopback interface (all sharing the same localhost IP address). This means that there is no packet loss and practically no delay.

To set up a network, it makes use of very simple network configuration files. These exist of snippets of Python code that instantiate a list of Node definition objects and assigns a type to each Node, such as client or relay. A simple example can be seen in Figure 7.3. Depending on the type, Chutney picks certain template files and combines them to generate a `torrc` configuration file for each of the nodes. These template files contain snippets of `torrc` configuration options that get set based on the properties set to the Node definition objects. Because the network files are run as Python code, it is possible to add custom code to modify certain properties of the Nodes, such as setting custom variables on them.

```
# By default, Authorities are not configured as exits
Authority = Node(tag="a", authority=1, relay=1, torrc="authority.tmpl")
ExitRelay = Node(tag="r", relay=1, exit=1, torrc="relay.tmpl")
Client = Node(tag="c", client=1, torrc="client.tmpl")

NODES = Authority.getN(3) + ExitRelay.getN(5) + Client.getN(2)

ConfigureNodes(NODES)
```

Figure 7.3: A basic network definition as used by Chutney. In this case, a network with 3 directory authority nodes, 5 exit nodes and 2 client nodes is defined and configured. Chutney does not have any concept of topology.

Once all the Tor instance have their own customized configuration file generated, each of the Tor instances is launched. Depending on how Chutney is used, it has the option to wait for the local Tor network to bootstrap by watching the log files of each of the Tor instances. The network test step in Chutney is called the *verification* step and is limited to one single action that is the same for all of the Tor clients. It launches a single local server, called `EchoServer`, that takes TCP traffic on a single port and immediately returns a copy of the received data back to the client. For each of the client Nodes defined in the network file, a local SOCKS connection is created. This SOCKS connection has two steps: For the `send-data` step it connects to the local SOCKS port of the corresponding Tor client and asks to connect, trough the Tor network, to the `EchoServer`. Once the connection is successful, it sends a predefined amount of random data to the `EchoServer`. The exact random data and its length is a global value shared by all clients. Once all the data has been successfully outputted on the SOCKS connection, the `send-data` step is marked as complete. The second step is the `check` step. This step runs for any data that is received on the SOCKS connection. For every chunk of bytes

that is received, it is checked against the random data that has already been sent and that chunk of bytes is marked as received. Once all the data has been marked as received, the `check` step is also marked as complete. A schematic view of this process can be found in Figure 7.4.

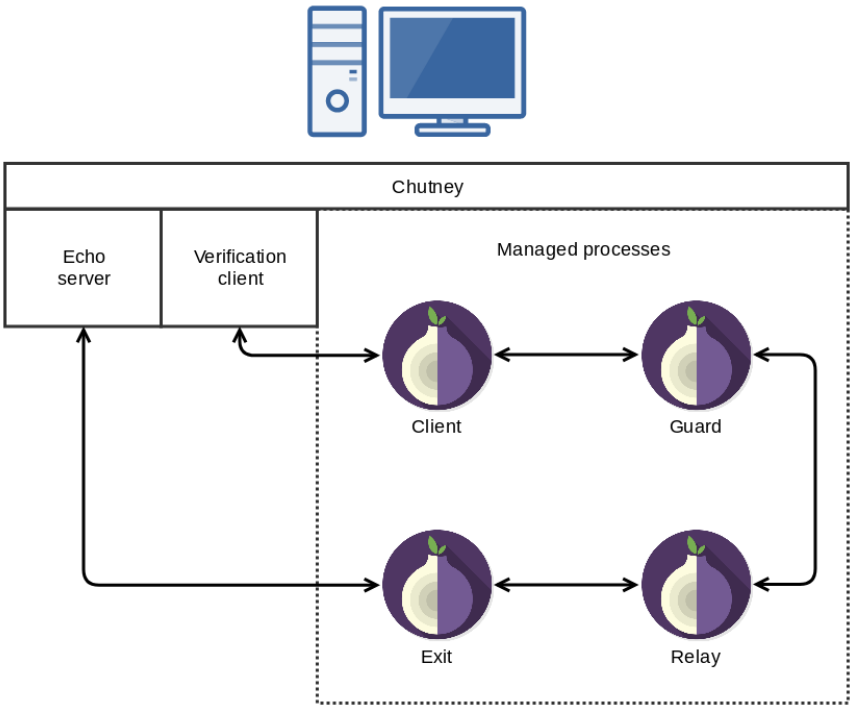


Figure 7.4: A schematic view of the Chutney architecture. The Chutney process configures and launches unmodified Tor binaries and waits for them to bootstrap. All Tor instances bind to localhost and all traffic goes through the local loopback interface. The verification clients connects to the Tor client and builds a circuit through the other Tor nodes, making a connection to the echo server. The verification client then sends random data, which the echo server receives and sends back, where the verification client verifies that all and the same data was returned by the echo server.

If both steps are marked as complete for each of the clients within a certain timeout, the verification step is considered successful. If one or more of the steps fails, the whole verification round might be repeated a predefined number of times until it is successful. Chutney does not keep or report any metrics about the performance of the tests, other than the speed of the slowest stream and the overall cumulative bandwidth of the whole network.

7.1.3 NetMirage

NetMirage [42] is a network emulator for large scale networks. It differs from Shadow in that it uses the real in-kernel network stack and does not require the tested applications to be modified or write application specific plugins to work with NetMirage. Because it presents itself as real, local network interfaces, it can work with existing, unmodified binaries.

To setup the virtual network it uses two types of nodes: the core node and one or more edge nodes. The core node is responsible for emulating layer 2 of the virtual network. To define this virtual network, NetMirage uses the same GraphML network topology files that shadow uses. These topology files are loaded by the core node and based on the defined nodes a number of Linux network namespaces are created. Network namespaces are a whole virtual network stack with their own network interfaces, IP addresses, network sockets, routing tables and more. They can be linked to other network namespaces by creating a virtual network interface in each of the namespaces and linking those together. To simulate various network conditions such as a limited amount of bandwidth, network latency or packet loss, NetMirage uses Open vSwitch [15]. Open vSwitch is a Software Defined Networking (SDN) virtual switch that can do layer 2 routing and bridge network interfaces together.

The edge node is responsible for emulating layer 3 of the virtual network and for running the actual applications. It does so by connecting to the core node an by allocating a list of IP addresses in a specified IP subnet. Each of the allocated IP addresses gets its own virtual network interface. Each application has to be configured to bind to one of the network interfaces. To run Tor network experiments, a slightly modified version of Chutney is used to launch the Tor processes, where each of the defined Tor nodes in the network file (such as in Figure 7.3) is assigned an IP address to bind to.

Network traffic that is destined for any of the edge node's virtual interfaces is routed from the edge note to the core node and to the core's root network namespace. The root network namespace routes the traffic to the network namespace corresponding to the appropriate node as defined in the topology file, where network limitations such as packet loss, bandwidth and latency are applied using Traffic Control (see Section 7.2). The traffic is then routed back to the edge node trough the core node's root network namespace. A schematic overview of the setup can be found in Figure 7.5.

The core node and edge nodes have to run on separate machines and are to be linked by a dedicated network link. This puts a limit on the ease of use of NetMirage, as it requires two separate machines with a dedicated network link between them in addition to their normal network interface. It also requires root access on both the core and edge nodes, as it manipulates the hosts' network stack.

However, during the evaluation it came to light that while the unmodified TCP-based Tor client worked well in NetMirage, the modified QUIC-based Tor client did not. Even with the simplest topology where all Tor clients were connected to

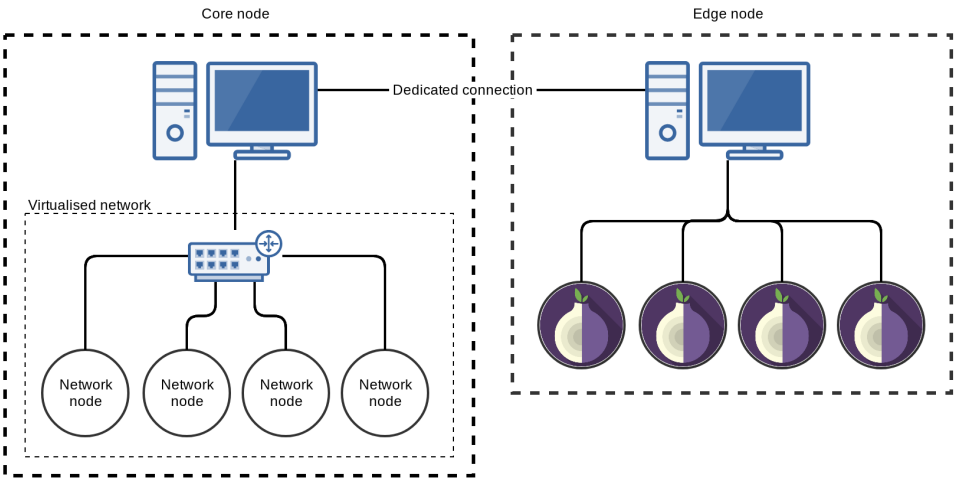


Figure 7.5: Schematic view of the NetMirage setup topology, consisting of two nodes that are connected via a dedicated network connection. On the core node, the specified GraphML network topology is simulated in a virtualized network. The edge node runs the Tor processes, that get assigned a dedicated IP address from the virtualized network. Any traffic to these IP addresses is forwarded over the dedicated connection to the core node, routed over the virtualized network, and then send to the edge node.

a single point in the graph (like in Figure 7.2) without any artificial loss or delay, the QUIC-based Tor client often failed to bootstrap the network or experienced long, unexplained delays in communication. Whether this is a limitation of NetMirage, the implementation of the QUIC-based Tor client or a combination thereof is unfortunately not known due to limited time constraints.

7.2 Manual network simulation

Because both Shadow and NetMirage did not work for their own reasons to evaluate the differences between the TCP-based Tor implementation and the QUIC-based Tor implementation, a manual virtual network setup approach was taken which in concept is similar to that of NetMirage, but without the problems that were encountered while evaluating the QUIC-based implementation as described in Section 7.1.3. In the following subsections, the individual parts that make up the evaluation setup are detailed. A schematic view of the resulting this setup can be found in Figure 7.6.

7.2.1 Network namespaces

As noted in section Section 7.1.3, the Linux kernel implements the concept of network namespaces, which are a whole virtual network stack with their own net-

work interfaces, IP addresses, network sockets, routing tables and more. The following network namespaces are used to emulate the virtual network: a general bridge namespace `ns-bridge`, a chutney namespace `ns-chutney` and a Tor node namespace `ns-`n for each of the Tor nodes that participate in the network, where n is the number of the specific Tor node. Each of the Tor node namespaces is linked to the `ns-bridge` namespace and the `ns-chutney` namespace by a linked virtual network interface pair, where the virtual interface in the Tor node namespaces gets assigned its own IP address based on the number of the node. A local route is added to make all the traffic that matches the subnet of the assigned IP address route through the associated virtual network interface. In the `ns-bridge` namespace, all virtual network interfaces linked to a Tor node interface are bridged together. This assures that all Tor node namespaces can route traffic to each other under the default network limitations, such as network latency. Using a bridge namespace like this also prevents the need to create n^2 linked network interfaces to make sure all Tor node namespaces can communicate with all other Tor node namespaces.

7.2.2 Traffic control

Traffic control is a framework in the Linux kernel that allows network traffic to be shaped or policies to be set on incoming traffic and traffic to be dropped. In this case, the network emulation layer, or `tc-netem` is of particular interest. It can for example be used to add artificial delays to local network interfaces to simulate network latency, add packet loss or rate limit network links.

The traffic routed through the `ns-bridge` namespace will have default network limitation applied to them, which means that all traffic between nodes will share the same limitations. To set custom limitations on a network link between two specific Tor node namespaces, an extra pair of linked virtual network interfaces can be created between them with a higher priority network route.

The virtual network links between the Tor node namespaces and the `ns-chutney` namespace have no additional network limitations applied as they emulate a local SOCKS connection between an application and a local Tor client.

7.2.3 Chutney

To set up a local test instance of the Tor network a modified version of Chutney was used. Chutney itself is launched in the `ns-chutney` namespace and each of the Tor processes launched by Chutney is then launched in its own Tor node network namespace. The network configuration files and the `torrc` templates have been modified so that each of the Tor nodes knows which IP address it needs to bind to. Because each of the Tor processes is running in its own network namespace, all traffic is routed through the virtual interfaces and the `ns-bridge` namespace where the network limitations are applied. Because the interfaces between the Tor node namespaces and the `ns-chutney` namespace do not have any network lim-

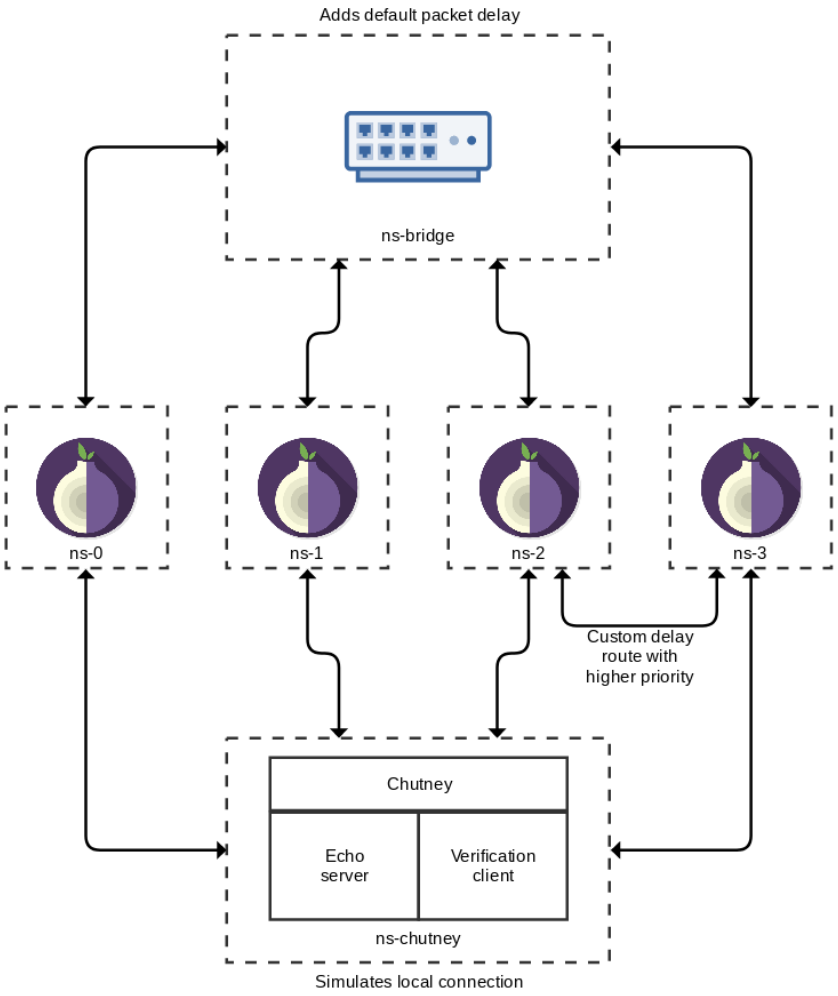


Figure 7.6: Schematic overview of the manual topology with the use of Linux namespaces. Chutney is launched in its own `ns-chutney` namespace, and launches each of the Tor processes in their own namespace `ns-`n. These namespaces are linked and bridged inside the `ns-chutney` namespace, connecting to the Chutney process. Another namespace, `ns-bridge` is also linked to all Tor processes where they are all bridged together. However, traffic through this namespace will have a delay added, simulating a real network. This is the default route the traffic between the Tor processes will take. Additional direct links between Tor namespaces can be make with a higher routing priority to add a custom delay between two Tor instances.

itations applied to them, this emulates the near-instant local connections between an application and a local Tor client.

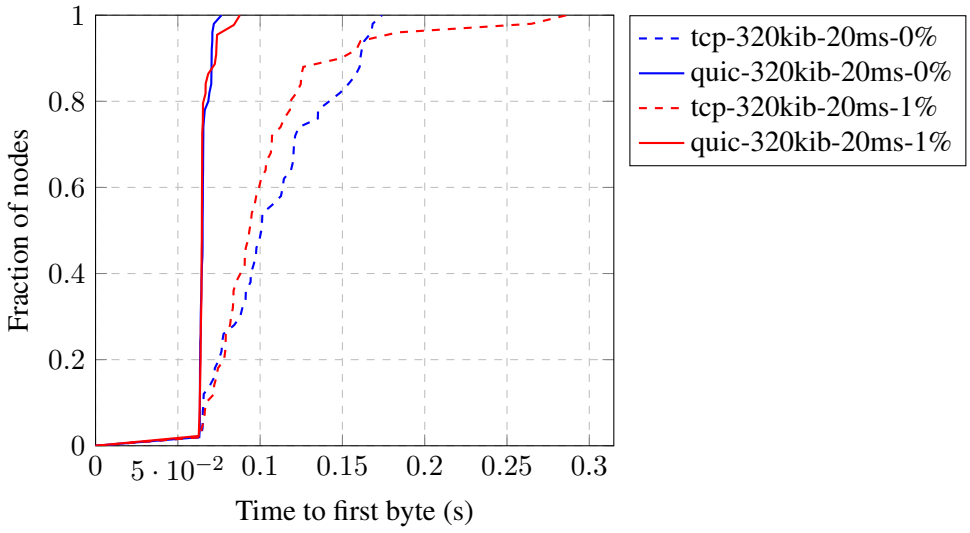
To collect the metrics of each of the experiments, Chutney had to be modified to record the timestamp of when the local SOCKS connection was initiated, the local SOCKS connection was established, when the first byte was received on the SOCKS connection and when the last byte was received on the SOCKS connection. With these timestamps, the *time to first byte*, the *time to last byte* and the *throughput* of each of the experiments is calculated and logged to a CSV file. This CSV file is then analysed at the end of the experiments to produce the cumulative metrics as shown in the graphs for each of the experiments.

Other modification to Chutney include the ability to differentiate in the amount of data each of the clients needs to transmit and the ability to watch the Tor processes while running an experiment to detect crashing Tor nodes.

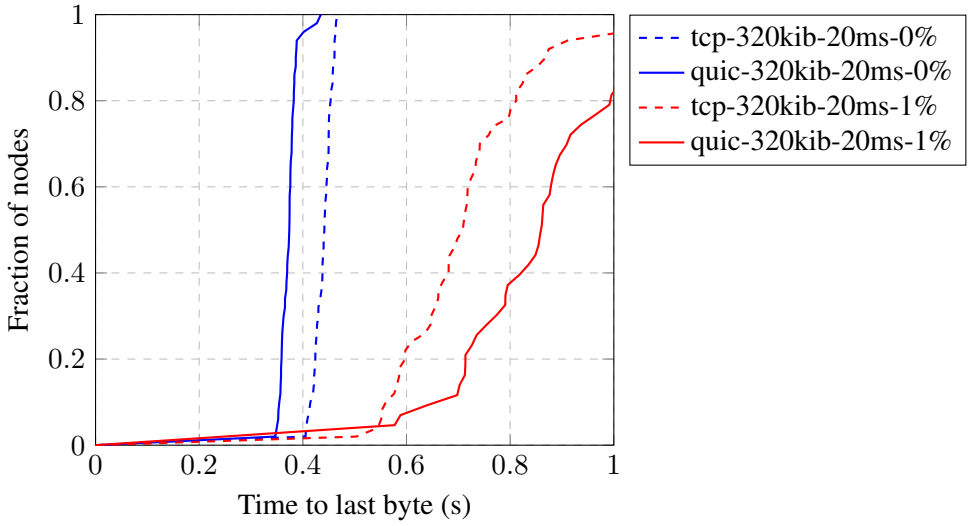
7.2.4 Limitations

However, due to the limitations of traffic control, only the latency of the network links can be emulated. As noted in the documentation of `tc`, packet loss on TCP connections gets reported back up instead of having to wait for a timeout to occur without packages being acknowledged by the receiving host as would be the case in a real network, which gives an obvious advantage to the TCP-based implementation of Tor over the QUIC-based implementation which uses UDP with packet loss detection instead implemented in user space [14]. The effects of this can be seen in Figure 7.7, which shows the performance metrics for the smallest network possible with one client OP_1, one guard node OR_1, one relay node OR_2, one exit node OR_3 and only a single possible circuit. A more detailed description of this scenario can be found in Section 8.2 and Figure 8.1.

Recall the performance metrics as defined in Section 5.1.3. Where the QUIC-based implementation performs equally well with or without packet loss in the time to first byte metric (where packet loss is unlikely to have an effect) and clearly beats the TCP-based implementation as expected, it performs significantly worse with packet loss compared to the TCP-based implementation in the time to last byte metric, against expectations, because of the unfair advantage that the TCP-based implementation has.



(a) Time to first byte



(b) Time to last byte.

Figure 7.7: The time to first byte and last byte comparison between the TCP-based and the QUIC-based implementations, on a transfer of 320 KiB with a 20 ms per hop link delay and either a 0% or 1% per hop packet loss. This demonstrates the shortcomings of using `tc` to artificially simulate packet loss on a local connection.

Chapter 8

Network performance evaluation

Testing the QUIC transport protocol implementation in Tor has been performed for several scenarios, varying from simple best and worst case scenarios to compare the theoretical performance effects, to a scaled down realistic model of the real Tor network to compare the practical performance effects. However, due to the limitations of the network simulation, only small networks could be tested.

8.1 Experimental setup

Recall the metrics from the performance design requirement (see Section 5.1.3) to evaluate the performance of the QUIC-based Tor prototype. In the used experimental setup, they are defined as follows:

- Time to first byte: This is defined as the amount of time between when the local SOCKS connection is established to the time the first byte is returned from the remote server. In this time, the client sets up a circuit, connects through the circuit via the exit node to the destination server, sends its first `DATA` cells and received the first `DATA` cell back from the exit node.

- Time to last byte: This is defined as the amount of time between when the local SOCKS connection is established to the time the last byte is returned from the remote server. In this time all the steps that are taken in the time to first byte metric are included, but now the client will have to send *all* its `DATA` cells and receive *all* its `DATA` cells back from the exit node.

For each scenario, there are two flavours: one with a default, vanilla Tor version 0.3.5 that uses the TCP transport protocol, and one with a modified Tor version 0.3.5 that uses the QUIC transport protocol. Both versions of Tor were modified to make it possible to preselect the guard, relay and exit node of a circuit for consistency reasons. This is also the only modification that was made to the TCP-based Tor version.

The network setup uses a manually defined combination of namespaces as detailed in Section 7.2. Where applicable, different amounts of latency were tested. The results for each of the scenarios and metrics are presented in graphs, where the dashed lines show the results for the TCP-based implementation, and the solid lines show the results for the QUIC-based implementation. Experiments that have the same parameters also have the same line colour in the same graph. For experiments with only a single or a small number of clients, the same experiment is repeated and the results of those experiments is combined until there are at least 50 data points for each specific metric. For experiments with a larger number of clients, the experiment is ran only once or a limited amount of times as they produce more data points per experiment. The legend labels have the format of {`protocol`}-{`payload_size`}-{`hop_latency`}-{`hop_loss`}, where:

- `protocol` is either `quic` for the modified QUIC-based implementation of Tor or `tcp` for the unmodified TCP-based implementation.
- `payload_size` is the size of the transfer payload, either 320 KiB or 5 MiB.
- `hop_latency` is the default per-hop network latency in milliseconds on the network link between two nodes, either 20 ms or 40 ms.
- `hop_loss` is the default per-hop percentage of packet loss on the network link between two nodes.

8.2 Scenario 1: A single circuit path with a single client

The first experiment is the smallest possible network: a single client OP_1 with a guard node OR_1, a relay node OR_2 and an exit node OR_3, as seen in Figure 8.1. This represents a best case circuit, as the client has all the bandwidth to itself. While the scenario is rather unrealistic, it is the simplest scenario possible and shows if the QUIC protocol has a positive or negative effect without outside influences.

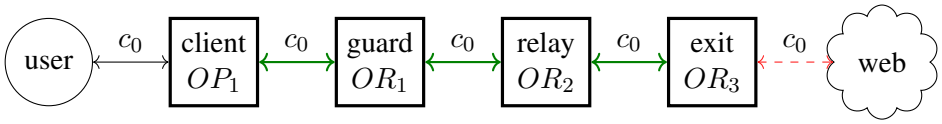


Figure 8.1: The setup of the first scenario. It includes a single client with a single circuit c_0, with nodes OR_1, OR_2 and OR_3.

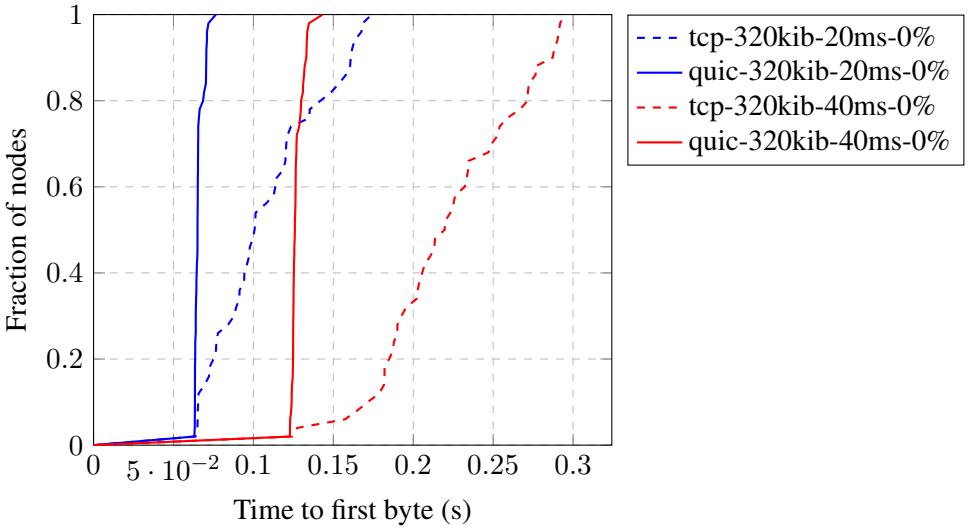
To evaluate the scenario two differently sized payloads will be used (320 KiB and 5MiB). Each individual link between each node has a latency of either 20 ms or 40 ms, bringing the round trip latency to respectively 120 ms and 240 ms. Because there is only one client in this scenario, it will be repeated 50 times. The result is presented as the fraction of nodes which have reached the time to first byte or the time to last byte.

8.2.1 Results and discussion

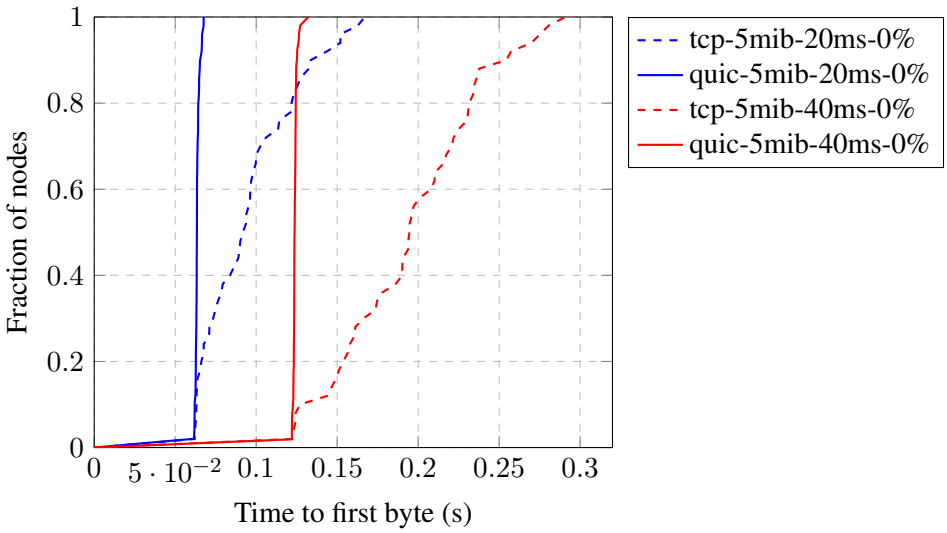
The results of the first scenario can be seen in Figures 8.2 and 8.3. As expected, it is clear that the QUIC-based implementation has an obvious advantage over the TCP-based implementation when looking at the time to first byte metric. Where the QUIC graphs show almost a vertical line where all clients get the first byte at nearly the same time, the TCP graphs show a much bigger variation in time between the initiation of the connection and when the first byte arrives on the client. An explanation for this can be that the TCP-based implementation uses the TCP slow start algorithm, while the QUIC-based implementation is much more aggressive in starting the connection. In addition to that, in the TCP-based implementation the TCP connection handshake and the TLS encryption handshake are separate, sequential processes, while the QUIC-based implementation combines those two in a single handshake (as described in Section 3.5.1). This appears to be the case for both the small transfer (Figure 8.2a) as the large transfer (Figure 8.2b).

Doubling the amount of network latency roughly doubles the time to first byte metric for the QUIC-based implementation, suggesting that there is no adverse effect other than the latency itself. For the TCP-based implementation however, doubling the latency roughly triples the time to first byte metric. This can again be explained by the additional and separate handshakes that are performed and which take multiple round trips.

For the time to last byte metric, the QUIC-based implementation again have an advantage over the TCP-based implementation, where the QUIC-based implementation is about 10% faster with a 20 ms network latency, and about 15% faster with a 40 ms network latency. As the transfers are run sequentially and have the full dedicated network link to themselves, there is no interfering traffic that can explain this difference and the advantage that the QUIC-based implementation has over the TCP-based implementation in the time to first byte metric is not significant enough to explain the difference. Likely, the QUIC-based implementation is using a more aggressive congestion control algorithm that allows for a slightly higher throughput than the standard TCP congestion control algorithm combined with more efficient acknowledgements that can also be tacked onto regular data packets, reducing the number of different packets sent. This appears to be the case for both the small transfer (Figure 8.3a) as the large transfer (Figure 8.3b).

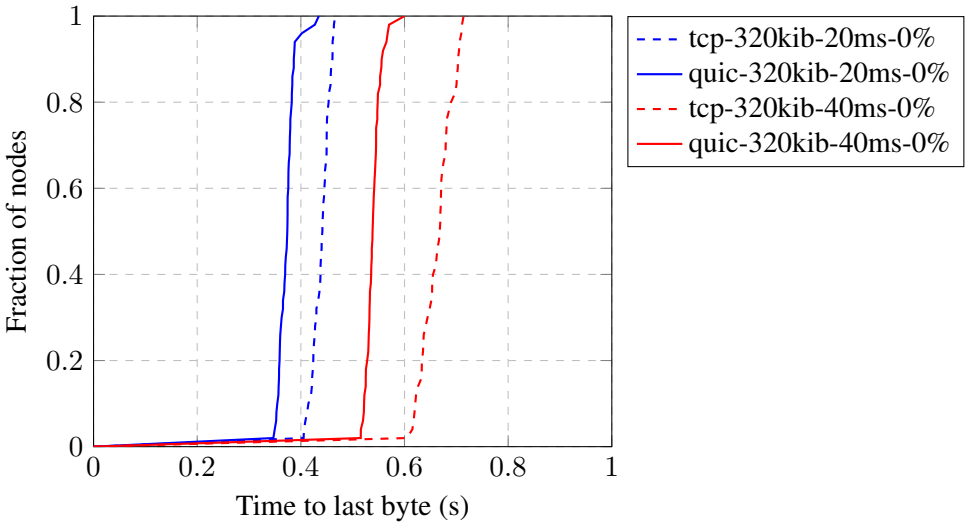


(a) Transfers of 320 KiB.

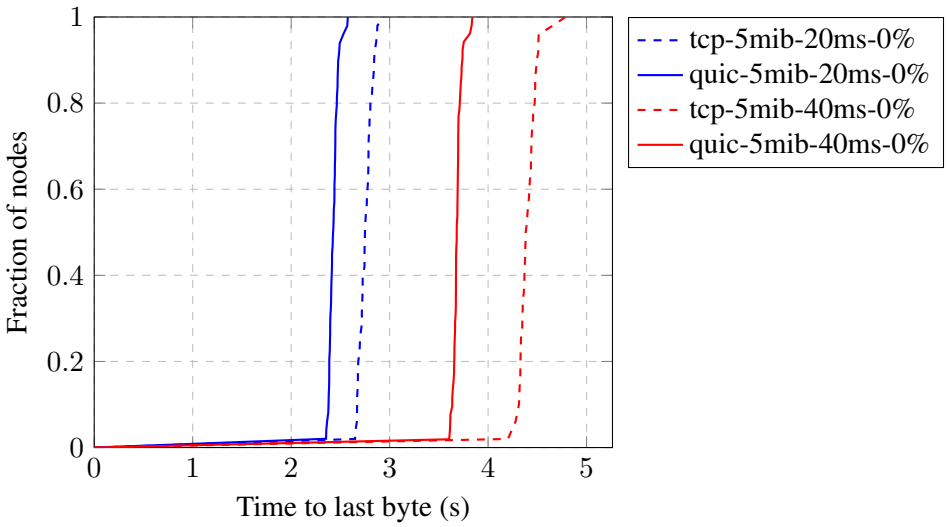


(b) Transfers of 5 MiB.

Figure 8.2: Scenario 1: the time to first byte by transfer size, with varying amounts of latency.



(a) Transfers of 320 KiB.



(b) Transfers of 5 MiB.

Figure 8.3: Scenario 1: the time to last byte by transfer size, with varying amounts of latency.

8.3 Scenario 2: Two circuit paths with two clients, sharing a single node

The second scenario has a distinct circuit path for each of the clients with their own nodes, except for a single node, as seen in Figure 8.4. Since this means that the two circuits do not share any connection between the nodes, the expected result is that there should not be any significant difference between QUIC-based implementation and plain TCP-based implementation, as independent connections should not suffer from the head-of-line problem and do not share a TCP congestion window. The only real shared limitation in this scenario is the bandwidth and the local resources of the shared node OR_3.

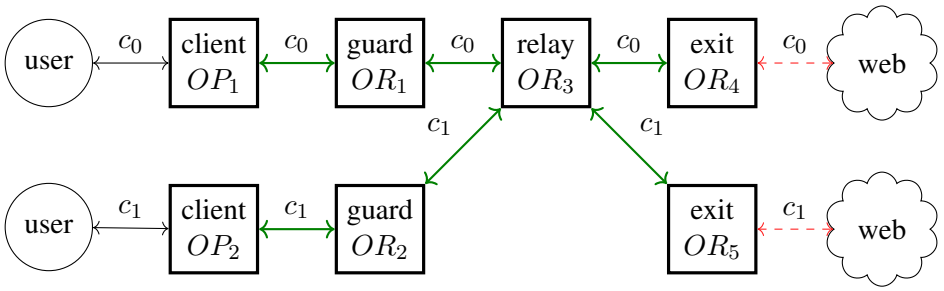
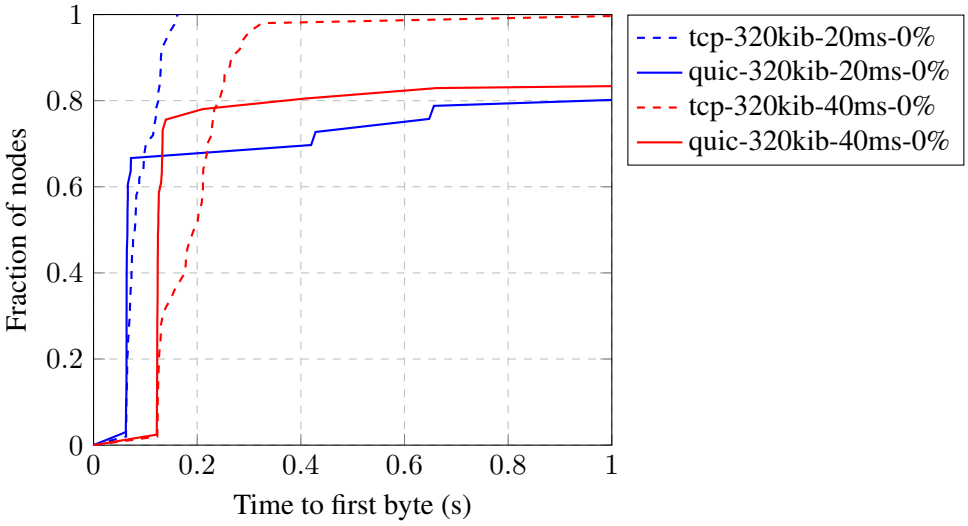


Figure 8.4: The setup of the second scenario. It includes two clients whose circuits c_0 and c_1 share only a single node, which means they do not share any connection.

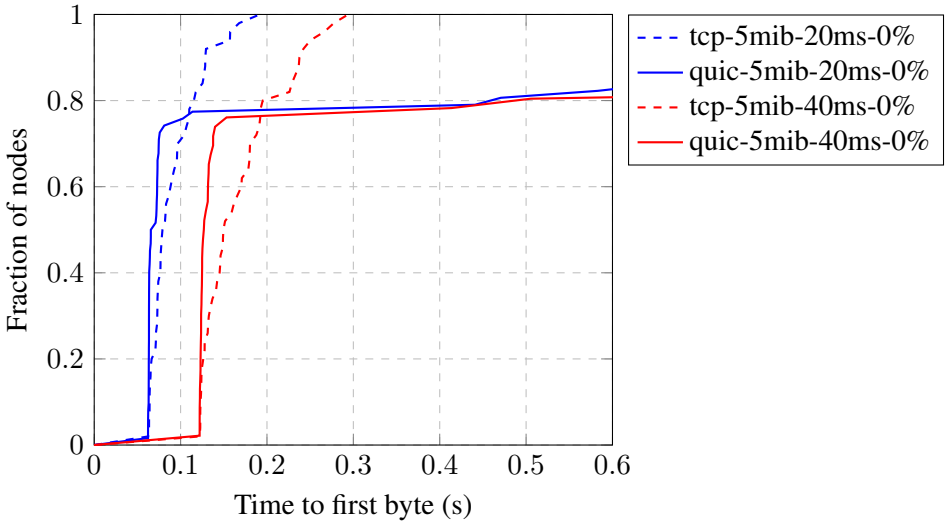
8.3.1 Results and discussion

The results of the second scenario can be seen in Figures 8.5 and 8.6. When looking at the time to first byte graph, the lines look similar to that of the first scenario with the QUIC-based implementation having about the same advantage over the TCP-based implementation, until it gets to around 70% to 80% of the clients where the line for the QUIC-based implementation suddenly becomes almost horizontal. It appears that some of the clients stall during the formation of a circuit. This is not necessarily due to the two circuits interfering on a network level. Because the experiment only has two clients and thus only returns two distinct data points for each round, the experiment was repeated 25 times in order to gain 50 distinct data points. For some rounds, both clients fell into the first 70% of clients, for some rounds both clients stalled, and for some rounds one of either clients stalled. The distribution of this can is show in Table 8.1 for each of the different combinations. Since both clients *only* share the relay node OR_3, but no network links, the issue must occur inside the relay node. Upon further inspection, the issue is likely a

scheduling issue within OR_3[1]. However, due to time constraints it was not possible to effectively fix this issue before running the evaluation.



(a) Transfers of 320 KiB.



(b) Transfers of 5 MiB.

Figure 8.5: Scenario 2: the time to first byte by transfer size, with varying amounts of latency.

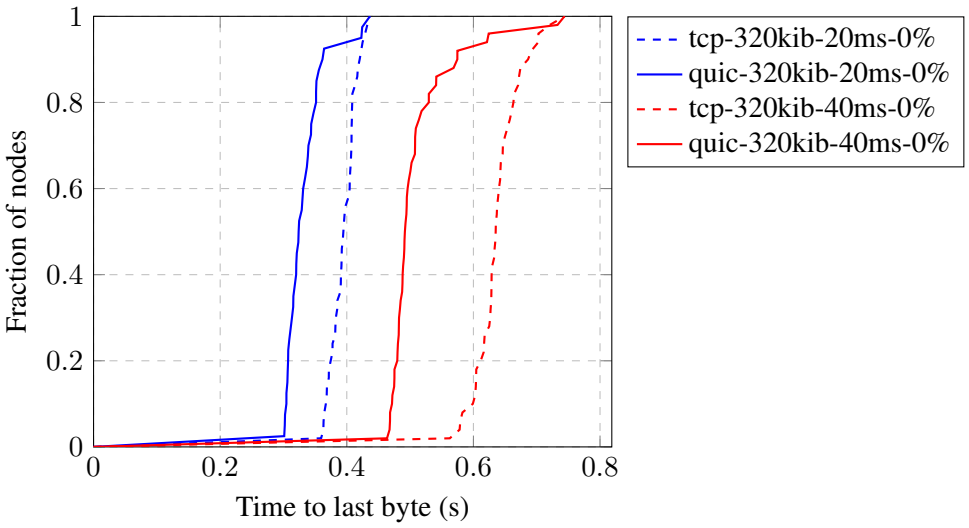
The stalling issue mostly effects the circuit creation, and not the data transfer

[1]Recall the sequence diagram in Figure 6.2, in particular the read connection block. It appears that the listener socket does read the incoming data and finds a matching connection in a time frame comparable to the TCP-based version. However, the time to fire the time specific read handler was not always consistent, occasionally taking a lot longer, which is a likely explanation of the stalling.

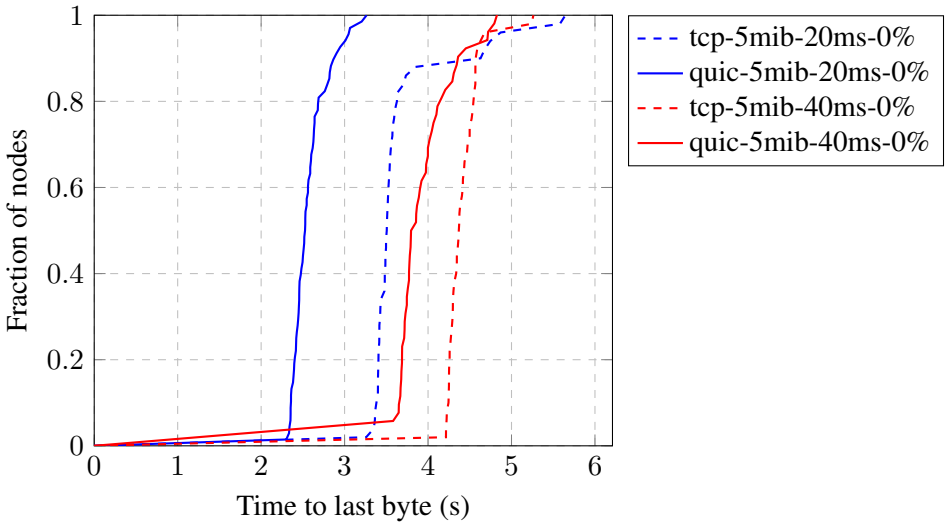
	Hop delay	
# Stalled clients	20 ms	40 ms
0 of 2	65%	56%
1 of 2	26%	40%
2 of 2	9%	4%

Table 8.1: Scenario 2: The distribution between the number of stalled and non-stalled clients for each round for the time to first byte metric with the QUIC-based implementation of the Tor client. The percentages reflect the fraction of rounds in which that number of nodes stalled. The data suggests that the client's circuits are not directly interfering and are not directly making the other client stall. In total, in both case about 20% to 25% of the clients initially stall. Thus, the fraction of clients that did not stall is about 75% to 80%.

itself. As can be seen in Figure 8.6a where the time to last byte metric is graphed, the results look very similar to the first scenario for more than 90% of the QUIC-based implementation clients. Only less than 10% of the clients somewhat slower than most other QUIC-based clients, although they still perform at least as well or slightly better than the TCP-based implementation.



(a) Transfers of 320 KiB.



(b) Transfers of 5 MiB.

Figure 8.6: Scenario 2: the time to last byte by transfer size, with varying amounts of latency.

8.4 Scenario 3: A single circuit path with two interfering clients

The third scenario is a variation on the first one, but adds an additional client, as seen in Figure 8.7. Both clients are using the same nodes in their circuit and thus will share the same connection between the nodes. Client 1 will have a low-latency, low-bandwidth load (such as is typical with web browsing) while client 2

will have a high-latency, high-bandwidth load (such as is typical with downloading large files). This demonstrates the classic worst-case scenario for the head-of-line blocking problem.

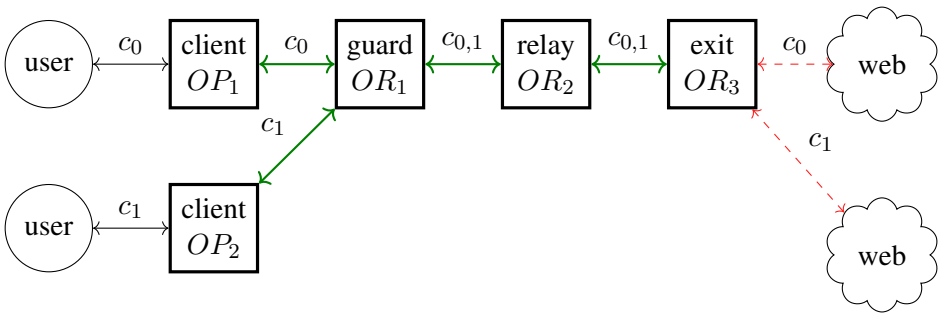


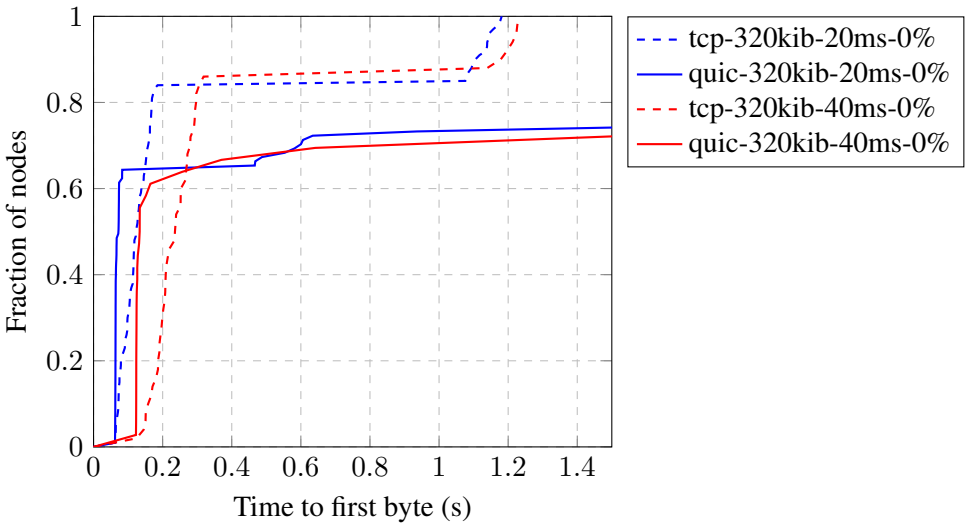
Figure 8.7: The setup of the third scenario. It includes two clients whose circuits c_0 and c_1 share the same nodes and thus connections.

8.4.1 Results and discussion

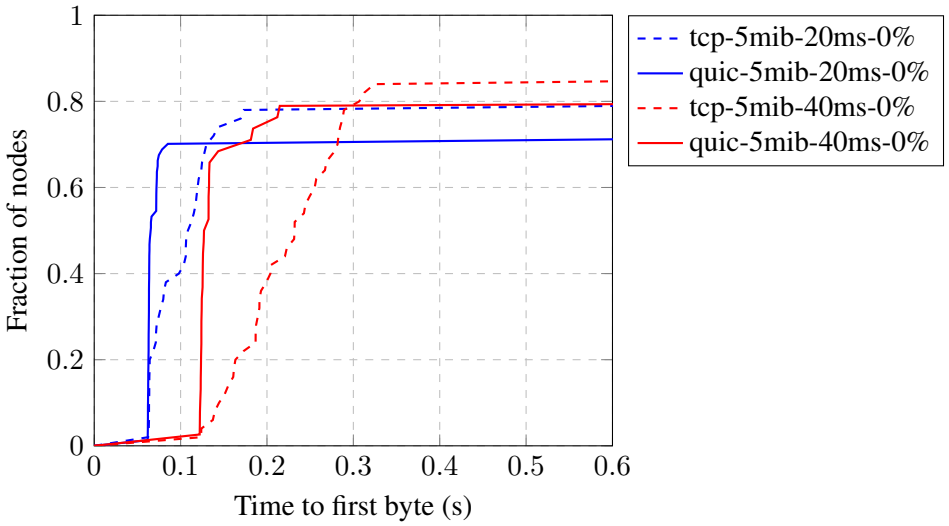
The results of the second scenario can be seen in Figures 8.8 and 8.9. When looking at the time to first byte graph, the lines look similar to that of the second scenario with the QUIC-based implementation having about the same advantage over the TCP-based implementation, until it gets to between 70% and 80% of the clients where line for the QUIC-based implementation suddenly becomes almost horizontal again. Again, it appears that some of the clients stall during the formation of a circuit. However, in comparison to the previous scenario, the TCP-based implementation also stalls, although only slightly later when it gets to between 80% and 85% of the clients. Again, as can be seen in Table 8.2, there does not necessarily appears to be a direct interference between the two clients as there is no significant difference between this scenario and the previous scenario, with latency having a bigger influence on the TCP-based clients than the QUIC-based clients.

For the TCP-based implementation however, there is a significant number of clients stalling where there were no stalling clients in the previous scenario. Seeing as the difference is that the two clients now share the same nodes in the circuit instead of having separate network links in their circuits, this suggest that the two circuits are interfering with each other while they are getting created.

Again, the stalling issue mostly effects the circuit creation, and not the data transfer itself. As can be seen in Figure 8.9a where the time to last byte metric is graphed for the 320 KiB transfer, the results look very similar to the earlier scenarios for about 85% of QUIC-based implementation clients with 20 ms link latency. Interestingly, the clients with 40 ms of link latency do not show any stalling, although the line is less horizontal and follows the curve of the TCP-based implementation more than in the previous scenarios. The clients with a 5 MiB transfer (Figure 8.9a) show a similar pattern, although a larger fraction stalls. Again, the



(a) Transfers of 320 KiB.



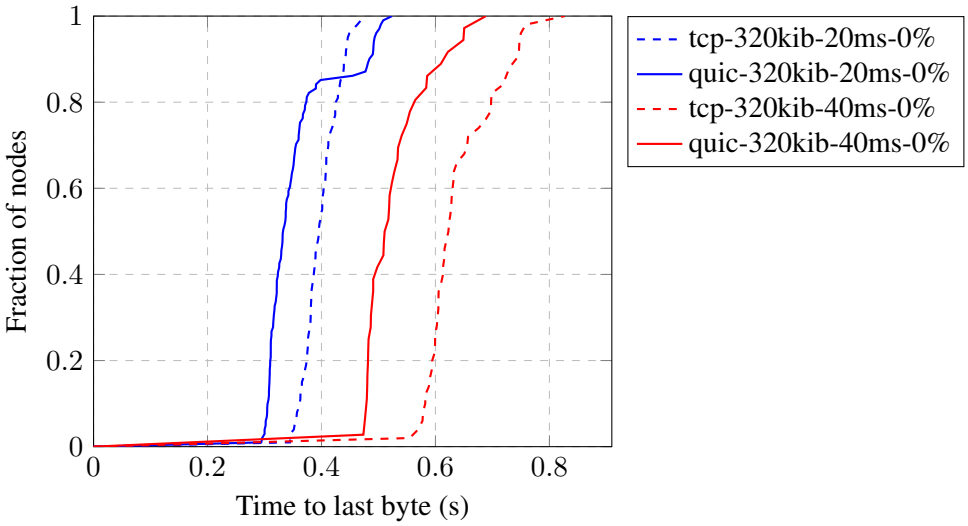
(b) Transfers of 5 MiB.

Figure 8.8: Scenario 3: the time to first byte by transfer size, with varying amounts of latency.

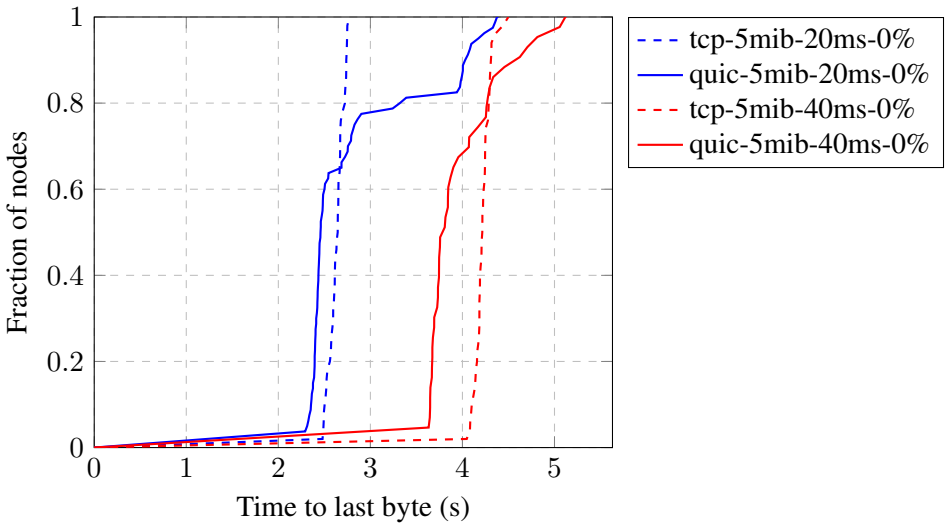
QUIC-based implementations are faster than the TCP-based implementation for most or all of the clients in the first case and a clear majority in the last case. The difference increases in favour of the QUIC-based implementation as the latency increases.

	Hop delay	
# Stalled clients	20 ms	40 ms
0 of 2	52%	47%
1 of 2	42%	46%
2 of 2	6%	7%

Table 8.2: Scenario 3: The distribution between the number of stalled and non-stalled clients for each round for the time to first byte metric with the QUIC-based implementation of the Tor client. The percentages reflect the fraction of rounds in which that number of nodes stalled. The data suggests that the client's circuits are not directly interfering and are not directly making the other client stall. In total, in both case about 25% to 30% of the clients initially stall. Thus, the fraction of clients that did not stall is about 70% to 75%.



(a) Transfers of 320 KiB.



(b) Transfers of 5 MiB.

Figure 8.9: Scenario 3: the time to last byte by transfer size, with varying amounts of latency.

8.5 Scenario 4: Two circuit paths with two clients, sharing two nodes

The fourth scenario is a variation on the second and third scenario. There are still two clients with two circuit paths, but now those two circuit paths share two of their three nodes. This means that while the connections between the client and the first node and the first node and the second node are distinct from each other, the connection between the second and the third node is shared between the two circuits.

As with scenario 3, client 1 will have a low-latency, low-bandwidth load while client 2 will have a high-latency, high-bandwidth load to demonstrate the head-of-line blocking problem.

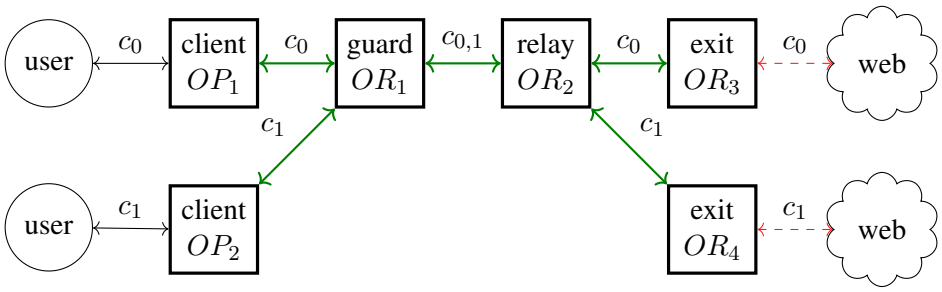
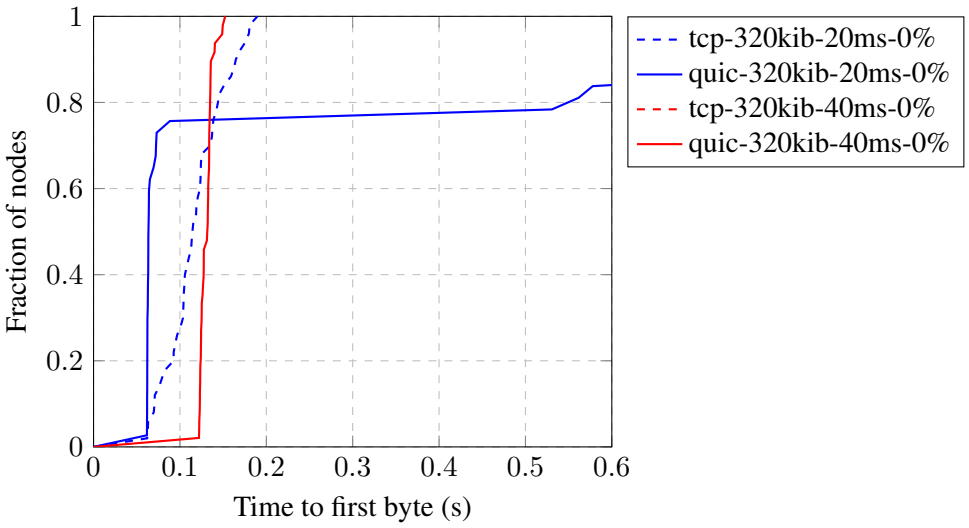


Figure 8.10: The setup of the fourth scenario. It includes two clients whose circuits share the same nodes and thus connections.

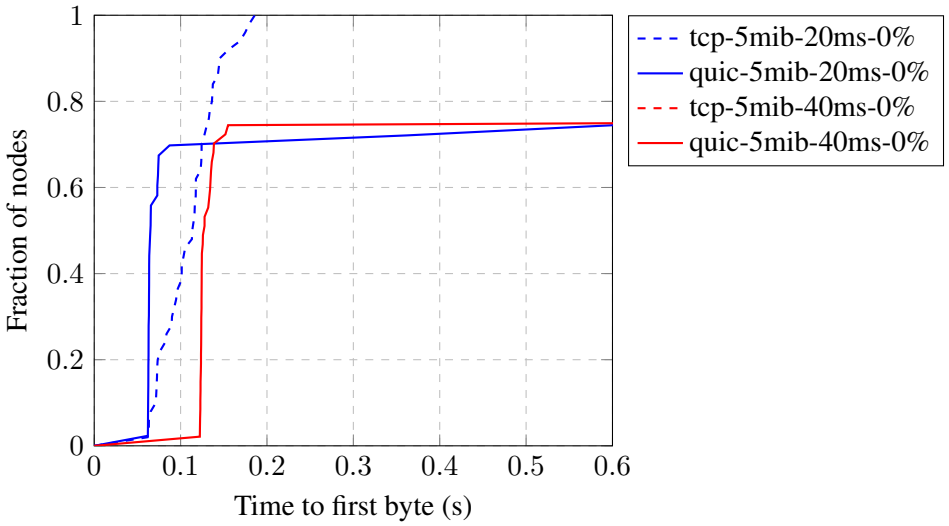
8.5.1 Results and discussion

The results of the fourth scenario can be seen in Figures 8.11 and 8.12. The TCP-based 40 ms delay transfers are not included in these graphs because the simulation failed to run for this specific case, despite numerous attempts. The Tor clients either failed to bootstrap or to open new circuits which made it impossible to run the transfers.

When looking at the time to first byte graph, the lines look similar to that of the second scenario, with the QUIC-based implementation having a nearly vertical line with the same advantage over the TCP-based implementation, until around 70% to 75% of the clients, when the lines become nearly horizontal. One interesting exception is the 320 KiB transfer clients under the 40 ms link delay, which does not have any stalling clients. The reason for this difference between the two transfer sizes is not exactly clear. As shown in the previous scenarios, the stalling appears to be a matter of chance instead of clients directly interfering with each other. The stall rates for this scenario, which can be seen in Table 8.3, again show the same behaviour. A likely explanation is that the 320 KiB clients got lucky and none of them stalled.



(a) Transfers of 320 KiB.



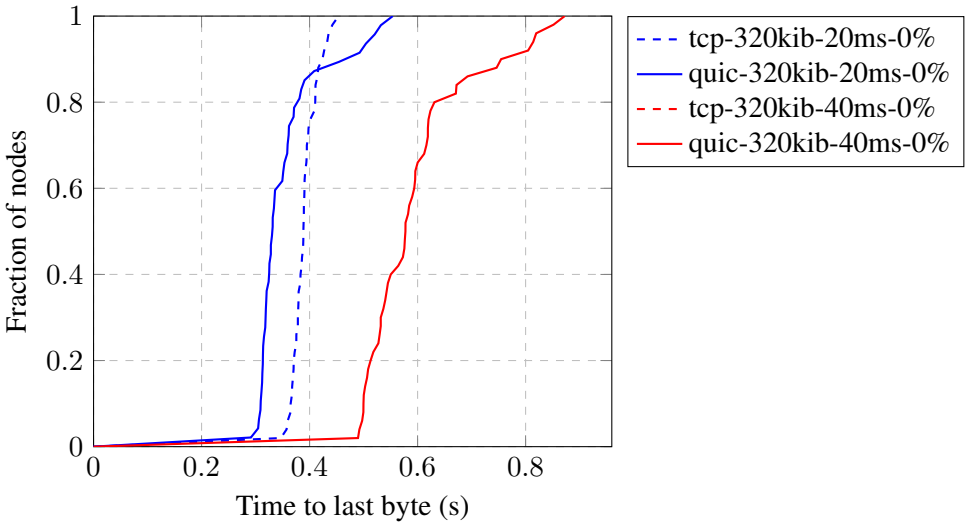
(b) Transfers of 5 MiB.

Figure 8.11: Scenario 4: the time to first byte by transfer size, with varying amounts of latency.

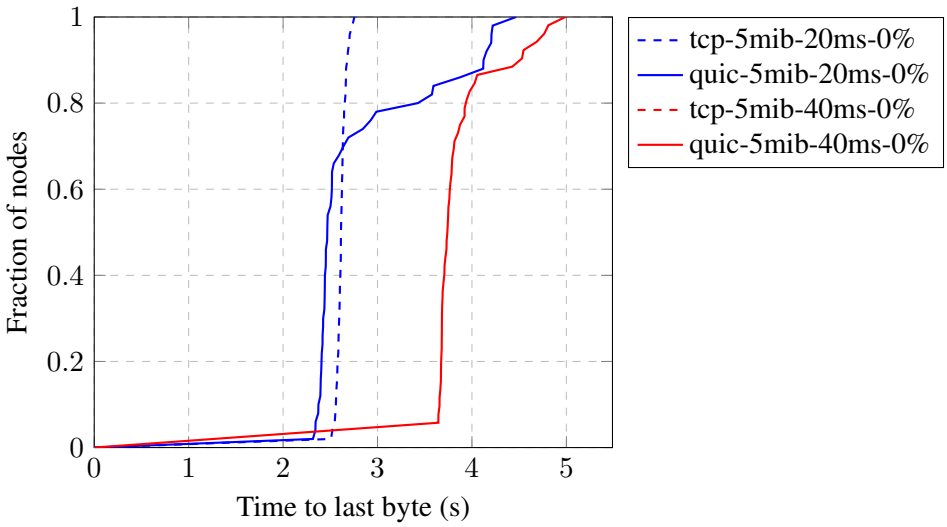
Similar to the previous scenario, the QUIC-based clients outperform the TCP-based clients until about 70% and 90% on the time to last byte metric, as seen in Figure 8.9a. There is no concise answer to give if the latency increase is still advantageous to the QUIC-based clients, because there is no data for the TCP-based clients.

	Hop delay	
# Stalled clients	20 ms	40 ms
0 of 2	49%	65%
1 of 2	45%	35%
2 of 2	6%	0%

Table 8.3: Scenario 4: The distribution between the number of stalled and non-stalled clients for each round for the time to first byte metric with the QUIC-based implementation of the Tor client. The percentages reflect the fraction of rounds in which that number of nodes stalled. The data suggests that the client's circuits are not directly interfering and are not directly making the other client stall. The fraction of clients that initially stall is less than 30% and 20% for the 20 ms and 40 ms delay, respectively. Thus, the fraction of clients that did not stall is over 70% and 80%, respectively.



(a) Transfers of 320 KiB.



(b) Transfers of 5 MiB.

Figure 8.12: Scenario 4: the time to last byte by transfer size, with varying amounts of latency.

8.6 Scenario 5: A scaled down Tor network model

The fifth scenario is a scaled down, reasonably accurate model of the Tor network. Since the real Tor network has over 6000 relay nodes with a combined advertised bandwidth of around 400Gbit/s and peaks with over 3.5 million users [48], running a test network with the same size and bandwidth is not feasible. Luckily, Jansen et al. [27] describes a methodology to create a reasonably accurate scaled down model the Tor network, that has successfully been used in practice such as for the evaluation of performance the KIST scheduler [29].

However, due to the limitations of the network simulation, only a small network could be tested. In addition, there were problems running the simulation with larger transfers for both the TCP-based and the QUIC-based Tor implementations, in a similar fashion as the TCP-based client in the previous scenario. As a result, this scenario was limited to only 320 KiB transfer sizes. This led to a network with 8 Tor relay nodes and 16 Tor clients and the simulation to be repeated 5 times to get a sufficient number of data points. In contrast to the previous scenarios, the nodes in a circuit are not hardcoded. Because of the higher number of clients, circuits are build randomly as they are in the real Tor network.

8.6.1 Results and discussion

The results of the fifth scenario can be seen in Figures 8.13 and 8.14. The time to first byte metric looks similar to the previous scenarios now that there are multiple clients building circuits and interfering with each other throughout the network, more similar to how the real network works. The QUIC-based clients keep their consistent advantage over the TCP-based clients in this metric, that grows with the amount of latency on the network links. Until they start to stall, that is, which the QUIC-based clients do 10 to 15 percent points before the TCP-based clients, which is consistent with the previous scenarios.

Looking at the time to last byte metric in Figure 8.14, the results are a bit different from the previous scenarios. While in the previous scenarios the QUIC-based transfers had a majority or sometimes all of the transfers finishing before their TCP-based counterparts, in this case the intersection of the two lines is already around 50%. It appears that the read event scheduling issue that is suspected to cause clients to stall or the performance to degrade might be more pronounced when there are multiple random circuits following different paths through the network as opposed to (parts of) the same path as in previous circuits.

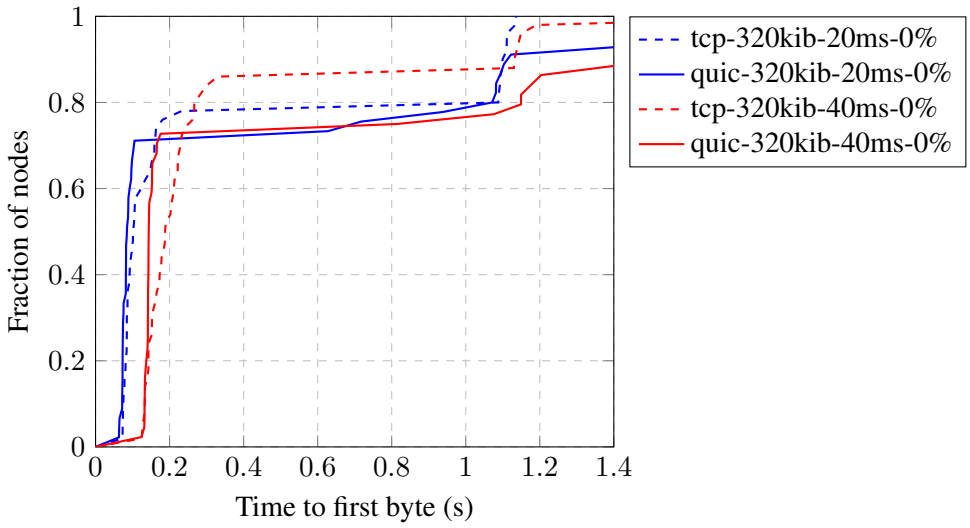


Figure 8.13: Scenario 5: the time to first byte with 320 KiB transfer size, with varying amounts of latency.

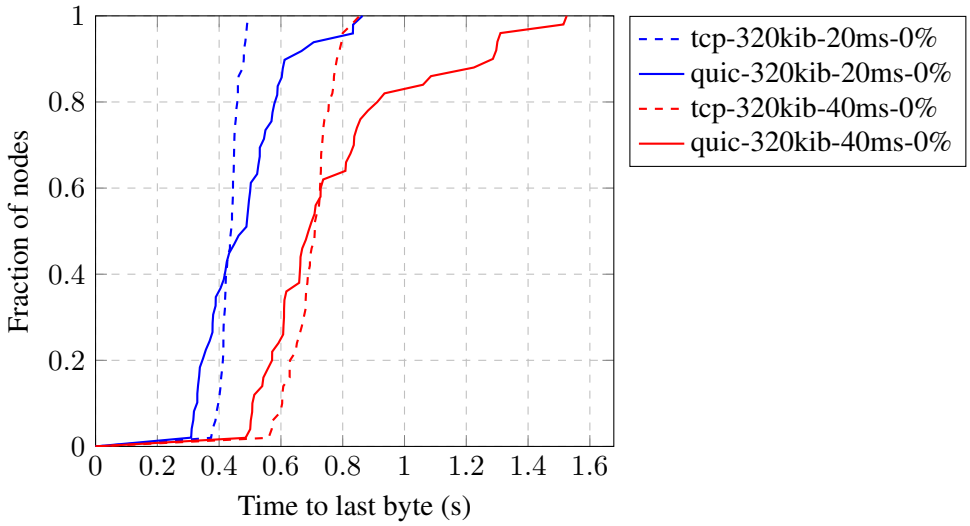


Figure 8.14: Scenario 5: the time to last byte with 320 KiB transfer size, with varying amounts of latency.

8.7 Discussion

Several reasons have been identified as to why the results are not as conclusive as expected:

- The network simulation: due to lacking support for UDP (Shadow) or less clear reasons (NetMirage), it was not possible to run the evaluation in a real

network simulator. Because it was not possible to either extend of fix the current simulators, a custom solution was developed based on Linux network namespaces and the Linux traffic control system. However, this solution is clearly limited. The biggest limitation is the inability to reliably simulate packet loss on TCP-based connections, a scenario is which QUIC is expected to have the advantage.

- The implementation: several instances have been observed where the performance decreased in cases where stalling was present with QUIC-based clients. The stalling is likely caused by a scheduling issue with the Tor read events (which Section 8.3.1 explains in more detail). This issue could not be resolved due to time constraints, despite researching efforts to identify and solve it. It is unknown how much the stalling exactly contributes to the performance degradation of the QUIC-based clients, and although they appear to be correlated, the underlying cause first needs to be solved in order for the correlations to become apparent.

- QUIC maturity: while it is claimed that the used QUIC library is used in production at Cloudflare [6], it is still under heavy development, as are all other QUIC libraries currently available. It might be that the way Tor uses the QUIC connections, such as sending lots of small cells over a stream, could cause the performance of the connection to be suboptimal.

With this in mind, the results do appear to indicate that sharing no circuit at all (scenario 1) gives the QUIC-based clients a clear performance advantage. As clients share the whole circuit, the stalling also happens with the TCP-based clients (scenario 3). However, these are just speculations based on the data available. There is no conclusive answer possible yet with the current data and more research is required.

Chapter 9

Future work

While this thesis lays the groundwork for the research into a QUIC-based transport protocol in Tor, in the following sections related additional future improvements are suggested.

9.1 Network performance evaluation

Partially because of the limitations of the network simulators, the evaluation of the prototype was limited. The already existing network simulators that were taken into consideration either had no support for the required network protocols (UDP) or had other limitations that made it impossible to use them. This is the reason that a manual, custom solution was developed based on Linux namespaces and the traffic control system, which had a more limited feature set than the existing network simulators. One example of such a limitation is that it was not possible to reliably test packet loss, which QUIC is expected to better deal with than TCP.

One options is to extend the network simulator that was used during the evaluation. One could route traffic over a real network before applying delays or package loss, similar to how NetMirage works. However, as this would be a significant investment of time, the required time is probably better spent improving the already existing network simulators. As Shadow is developed by a Tor developer and also actively used in Tor research, extending Shadow to work with UDP-based protocols instead of just TCP-based protocols is an avenue worth pursuing. Another options is to find out why NetMirage did not work with the UDP-based prototype. This is potentially less work, because it is not new implementation work as with Shadow, but the uncertainty of what the actual problem is makes it difficult to compare. Also, the pay off might be lower as NetMirage is not widely used yet (it is still beta software [42]), although that could change in the future, especially because NetMirage should work with *any* software, without modifications.

Using network simulators that were specifically built for and are know to work well with Tor also opens the doors to do large scale network tests with thousands of nodes.

9.2 Prototype implementation

The prototype of the QUIC-based is exactly that: a prototype. The code is not yet ready for general use or ready to be upstreamed as-is. One of the ways to get there, is to create a dedicated `QUICChannel` instead of modifying the `TLSChannel` to work with QUIC. However, this still needs a lot of work, because there are still a significant number of violations between the abstract `Channel` and the `TLSChannel` implementation[1].

9.2.1 TLS libraries

Another implementation improvement would be to stop mixing TLS libraries. The QUIC library that is used, Quiche, requires the BoringSSL library while Tor itself uses the OpenSSL library. As explained in Section 6.2.3, these do not mix well because BoringSSL is a fork of OpenSSL and still shares a lot of symbols with the latter. The best solution here is to make Quiche compatible with OpenSSL, which requires modifications to be upstreamed to the OpenSSL project. Coincidentally, during the last few days of this project, work was started to make Quiche work with a development version of OpenSSL[2]. Once this work is finished, both on the OpenSSL as on the Quiche side, this also opens the door to compiling Quiche statically into the Tor binary.

9.2.2 Backwards compatibility

Because of the hop-to-hop design, it is possible to mix both old TCP-based Tor instances and new QUIC-based Tor instances. Connections between nodes can automatically connect over QUIC if possible or fall back to TCP when needed. This requires Tor to listen on both a UDP and a TCP port. Ideally, this would be implemented through the `QUICChannel` and `TLSChannel` separation as described in the previous section. It would be interesting to evaluate the performance improvements in mixed networks where only part of the circuit uses QUIC.

9.3 Performance improvements

The performance of the prototype implementation was unfortunately not always optimal. Likely due to a scheduling issue of the read and write events of the QUIC-based connections, which causes cells to be delayed and some clients to stall. First and foremost, this needs to be confirmed and resolved to improve the performance.

Once that problem is fixed, there are still further improvements to the QUIC connection that could be considered. Does it for example make sense to use differ-

[1] As noted in this still open Tor ticket: `https://trac.torproject.org/projects/tor/ticket/23993`.

[2] See the relevant pull request: `https://github.com/cloudflare/quiche/pull/126`

ent types of congestion control algorithms for different kinds of loads on streams? If so, what are the security implications of this? Would an adversary be able to distinguish between different streams inside a circuit this way?

A second improvement is to implement Forward Error Correction (FEC). This is currently not yet implemented in Quiche, but might prove to be a performance improvement once it is.

A third improvement is to investigate the 0-RTT handshake of QUIC. Does it make sense to use this? What are the implications of using this (for example, losing perfect forward secrecy? Or losing replay detection?) and are they worth it compared to the possible performance benefits?

9.4 Security review

While the prototype design had no deep and big architectural changes to Tor network model, there needs to be a thorough security review of the implications of using a QUIC-based transport protocol instead of TCP and the way it is applied here. This to prevent mistakes like in the Quictor implementation (see Section 4.4).

Another issue is that lowering the latency inevitably makes it easier to do timing attacks against Tor [22, 38]. Research needs to be done to find out how big this impact is and if it is significant enough to need additional mitigations.

Chapter 10

Conclusion

This thesis has sought to answer the following research question:

> What are the effects on the performance of the Tor network when using a UDP-based protocol as the transport layer protocol?

To answer this question, first is was necessary to decide which UDP-based protocol would have to be used as the transport layer protocol. Previous research has already considered a few, such as the DTLS-based protocol in Section 4.2. However, the main issues with the current TCP-based protocol are very similar to the issues with the current HTTP/2 protocol as used on the web today. Considering that the QUIC protocol, also a UDP-based protocol, is specifically being developed to solve these issues, this was deemed to be the best available option.

The next step in answering this question was to define how to measure the performance import of different transport layer protocols in Tor. The two main metrics that were found to be important are the time between the when local SOCKS connection is established and the first byte is returned, or *time to first byte*, and the time between when the local SOCKS connection is established and the last byte is returned, or *time to last byte*. With these two metrics, the responsiveness and the throughput of the Tor circuits can be evaluated. Multiple Tor network simulators were evaluated but found lacking for either lack of UDP support or less clear reasons. A custom solution was developed in which and extended version of one of the existing tools was integrated. Section 7.2 explains this in more detail.

However, the final answer to the research question is as of yet not conclusive due to the limitations during the network performance evaluations. The QUIC-based implementation performs better on some metrics, mainly in the time to first byte metric on most scenarios, but only up to a point. Some clients stalled, likely due to a scheduling issue of the read handler, which would suggest that this is not caused by network traffic interference. However, without actually solving this issue, it is impossible to prove that this is the case. Additionally, the limitations with the existing network simulators had the consequence that no larger scale network simulation could be run. Without improving on these two limitations, it is impos-

sible to prove and therefore state with absolute confidence that using QUIC has a definitive performance benefit.

That being said, this thesis also has some other contributions that were made. It defines a design for a QUIC implementation and a prototype that implements that design. In addition to that, it contributes a script to set up virtualized network in which network delays can be simulated. The existing Chutney tool is modified to integrate with this, launching the Tor processes inside the virtualized network. Other modifications include logging individual metrics about the transfers, running multiple different transfers in parallel and monitor the Tor processes during the experiments.

With these contributions, this thesis has laid the groundwork for further research into the performance benefits of the QUIC transport protocol. It aims to inspire the continuation of the research towards a faster, widely used Tor. Privacy online should not be privilege, it is a universal human right.

Bibliography

[1] Mashael AlSabah and Ian Goldberg. Performance and security improvements for tor: A survey. 2015.

[2] Autoriteit Persoonsgegevens (AP). Dutch data protection authority: Facebook violates privacy law. https://autoriteitpersoonsgegevens.nl/en/news/dutch-data-protection-authority-facebook-violates-privacy-law.

[3] Autoriteit Persoonsgegens (AP). CBP issues sanction to google for infringements privacy policy. https://autoriteitpersoonsgegevens.nl/en/news/cbp-issues-sanction-google-infringements-privacy-policy, 2014.

[4] Chromium Blog. A quic update on googles experimental transport. https://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html, 2015.

[5] Buildswith. Web technology usage trends. https://trends.builtwith.com/analytics/, 2019.

[6] Cloudflare. Quiche readme. https://github.com/cloudflare/quiche/blob/master/README.md, 2019.

[7] Cisco CCENT/CCNA Networking Concepts. The osi network model - what you need to know. https://cisconetworkingbasics.blogspot.com/2013/06/the-osi-network-model-what-you-need-to.html, 2013.

[8] DARPA. Transmission control protocol. RFC 6793, 1981.

[9] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. 2004.

[10] Roger Dingledine and Steven Murdoch. Performance improvements on Tor. 2009.

[11] The Economist. The worlds most valuable resource is no longer oil, but data. https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data, 2017.

[12] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. 2016.

[13] Marc Fischlin and Felix Günther. Replay attacks on zero round-trip time: The case of the tls 1.3 handshake candidates. 2017.

[14] The Linux Foundation. Traffic control netem. https://wiki.linuxfoundation.org/networking/netem, 2019.

[15] The Linux Foundation. What is Open vSwitch? http://docs.openvswitch.org/en/latest/intro/what-is-ovs/, 2019.

[16] United States Federal Trace Commission (FTC). FTC imposes $5 billion penalty and sweeping new privacy restrictions on facebook. https://www.ftc.gov/news-events/press-releases/2019/07/

ftc-imposes-5-billion-penalty-sweeping-new-privacy-restrictions, 2019.

[17] Google. BoringSSL readme. https://boringssl.googlesource.com/boringssl/, 2019.

[18] IETF QUIC Working Group. QUIC: A UDP-based multiplexed and secure transport. https://tools.ietf.org/html/draft-ietf-quic-transport-22, 2019.

[19] IETF QUIC Working Group. QUIC implementations. https://github.com/quicwg/base-drafts/wiki/Implementations, 2019.

[20] The Guardian. Everyone is under surveillance now, says whistleblower Edward Snowden. https://www.theguardian.com/world/2014/may/03/everyone-is-under-surveillance-now-says-whistleblower-edward-snowden, 2014.

[21] Morley Gunderson and Byron Y Lee. Pay discrimination against persons with disabilities: Canadian evidence from pals. 2016.

[22] Nicholas Hopper, Eugene Y. Vasserman, and Eric Chan-Tin. How much anonymity does network latency leak? 2007.

[23] Freedom House. Freedom on the net 2018. https://freedomhouse.org/report/freedom-net/freedom-net-2018, 2018.

[24] United Kingdom Information Commissioner's Office (ICO). Ico issues maximum £ 500,000 fine to facebook for failing to protect users personal information. https://ico.org.uk/about-the-ico/news-and-events/news-and-blogs/2018/10/facebook-issued-with-maximum-500-000-fine/, 2018.

[25] J. Iyengar and M. Thomson. QUIC: A UDP-based multiplexed and secure transport (draft). https://quicwg.org/base-drafts/draft-ietf-quic-transport.html, 2019.

[26] Rob Jansen. Shadow wiki. https://github.com/shadow/shadow/wiki, 2019.

[27] Rob Jansen, Kevin Bauer, Nicholas Hopper, and Roger Dingledine. Methodically modeling the Tor network. In *Presented as part of the 5th Workshop on Cyber Security Experimentation and Test*, Bellevue, WA, 2012.

[28] Rob Jansen and Nicholas Hopper. Shadow: Running Tor in a box for accurate and efficient experimentation. 2012.

[29] Rob Jansen and Matthew Traudt. Tors been KIST: A case study of transitioning tor research to practice. 2017.

[30] C. Kiraly, G. Bianchi, and R. Lo Cigno. Solving performance issues in anonymization overlays with a L3 approach.

[31] Ron Kohavi and Roger Longbotham. Online experiments: Lessons learned. 2007.

[32] Kevin Ku and Xiaofan Li. QuicTor: Tor running on QUIC protocol. 2016.

[33] Adam Lerner, Anna Kornfeld Simpson, Tadayoshi Kohno, and Franziska Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. 2016.

[34] Greg Linden. Marissa Mayer at web 2.0. https://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html, 2006.

[35] Karsten Loesing, Steven J. Murdoch, and Roger Dingledine. A case study on measuring statistical data in the Tor anonymity network. In *Proceedings of the Workshop on Ethics in Computer Security Research (WECSR 2010)*, LNCS. Springer, January 2010.

[36] Isis Lovecruft, George Kadianakis, Ola Bini, and Nick Mathewson. Another algorithm for guard selection. https://gitweb.torproject.org/torspec.git/tree/proposals/271-another-guard-selection.txt, 2016.
[37] Elena Maris, Timothy Libert, and Jennifer Henrichsen. Tracking sex: The implications of widespread sexual data leakage and tracking on porn websites. 2019.
[38] Nick Mathewson and Mike Perry. Towards side channel analysis of datagram tor vs current tor.
[39] Damon McCoy, Kevin Bauer, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. Shining light in dark places: Understanding the Tor network. pages 63–76, 2008.
[40] Commission nationale de l'informatique et des liberts (CNIL). The CNILs restricted committee imposes a financial penalty of 50 million euros against google llc. https://www.cnil.fr/en/cnils-restricted-committee-imposes-financial-penalty-50-million-euros-again 2019.
[41] Michael F. Nowlan, David Isaac Wolinsky, and Bryan Ford. Reducing latency in tor circuits with unordered delivery. 2013.
[42] University of Waterloo CrySP. NetMirage. https://crysp.uwaterloo.ca/software/netmirage/, 2019.
[43] OrNetRadar. Monitoring the tor network for new relay groups and events. https://nusenu.github.io/OrNetRadar/, 2019.
[44] Frank Orozco. How quic speeds up all web applications. https://medium.com/@verizondigital/how-quic-speeds-up-all-web-applications-62964aadb3d1, 2018.
[45] J. Postel. User datagram protocol. RFC 768, 1980.
[46] The Tor Project. Chutney readme. https://gitweb.torproject.org/chutney.git/tree/README, 2019.
[47] The Tor Project. RustInTor - Tor bug tracker & wiki. https://trac.torproject.org/projects/tor/wiki/RustInTor, 2019.
[48] The Tor Project. Welcome to Tor metrics. https://metrics.torproject.org/, 2019.
[49] Joel Reardon. Improving Tor using a TCP-over-DTLS tunnel, 2008.
[50] Jim Roskind. QUIC: Design document and specification rationale. https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit, 2013.
[51] NMAP Network Scanner. Chapter 8. remote os detection. https://nmap.org/book/osdetect.html, 2019.
[52] Joost Schellevis and Winny de Jong. Verzekeraars sturen surfgedrag naar facebook, ook van medische pagina's. https://nos.nl/artikel/2226902-verzekeraars-sturen-surfgedrag-naar-facebook-ook-van-medische-pagin html, 2018.
[53] Florian Tschorsch and Björn Scheuermann. Mind the gap: Towards a backpressure-based transport protocol for the Tor network. 2016.
[54] Chris Wacek, Henry Tan, Kevin Bauer, and Micah Sherr. An empirical evaluation of relay selection in Tor, 2013.