QUUX: a QUIC un-multiplexing of the Tor relay transport

Ali Clark

supervised by
Dr. Steven J. Murdoch

September 1, 2016

Abstract

The Tor network currently multiplexes data for its circuits over a TCP connection for each relay pair. This work investigates the use of QUIC streams as an alternative transport design.

Acknowledgements

I would like to thank:

Steven for his exceptional patience as I frequently switched between dissertation topics, without which I might not have reached this one; and for his expert and thoughtful guidance throughout this project.

My sisters, Sophia for teaching me the invaluable tool of mindfulness-based stress reduction, and Venetia for her empathy and encouragement drawn from her own experience.

My parents for their love and support.

The many projects and their contributors without which this work would not be possible, in particular Chromium, libquic, Tor, Shadow, and Chutney.

Rob Jansen for his responsive and detailed help on Shadow issues.

Contents

1 Introduction **1**

1.1 Usability and security 2

2 Background **3**

2.1 The Internet 3

2.1.1 IPSec 4

2.2 Transport Protocols 4

2.2.1 UDP 4

2.2.2 TCP 4

2.2.3 TLS 6

2.2.4 DTLS 6

2.2.5 SCTP 6

2.2.6 QUIC 7

2.2.7 µTP 7

2.3 Tor 8

2.3.1 libevent 9

2.3.2 Relay data path 9

3 Related work **12**

3.1 Initiator-to-exit 13

3.2 End-to-end 13

3.3 Hop-by-hop 13

3.4 Conclusions 14

3.4.1 Network design 15

3.4.2 Transport protocols 15

4 Methodology 18
4.1 QUIC . . . 18
4.2 QUUX . . . 19
4.2.1 Linking . . . 19
4.2.2 Channels . . . 20
4.3 Chutney experiment . . . 21
4.4 Shadow experiment . . . 23

5 Results 24
5.1 Chutney experiment . . . 24
5.1.1 Bandwidth overhead . . . 25
5.1.2 Conclusions . . . 25
5.2 Shadow experiment . . . 27
5.2.1 Conclusions . . . 29

6 Conclusion 30
6.1 Backpressure . . . 30
6.2 Security . . . 31
6.2.1 Software . . . 31
6.2.2 Usability . . . 32

Appendix A quux.h 40

Appendix B Chutney experiment 45

Appendix C Shadow experiment 47

List of Figures

2.2 Queues in the Tor data path 10

4.1 Overview of the Chutney experiment 22

Chapter 1

Introduction

With the growing prominence of information technology in society, including private information as much as any other, privacy has become a defining issue of our time.

As a rule, people expect to have privacy in their affairs. Even for those who are accepting of information systems that have unexpectedly stored or processed their private data, often the logic isn't that they don't care about their privacy; instead it is that the storage or processing is in fact condoned within their view of privacy — either the data was not *really* private, or the other party wouldn't care to look at it anyway.

Since privacy is an issue of consent and autonomy over one's private information, we must not rely on these forms of retroactive and obligatory consent, lest they not be granted. For example, we can reasonably assume many victims of *LOVEINT* [Gor13] do not consent to the way their data was stored and processed, and have indeed been denied of their privacy.

As engineers, we must build information systems that allow the possibility of a person to retain control over their private information, and allow them the ability to give meaningful consent in privacy questions.

The internet as used for web browsing is an example where choice is needed. With each request a web server is made aware of its client's IP address, which will often identify the client to a fine-grained level that reveals their approximate location and provides an identifier that enables tracking across the web.

At the same time, it's near implausible to suggest someone refrain from using the web if they disagreed with this collection and processing of their private data. The Tor [DMS04] web browser provides a viable alternative. It implements an *overlay network* on top of the internet, so the final IP address accessing a web service is not the same as the client's IP address.

Tor reports over a million daily users [Met] and is the de facto research platform for academics of anonymous web browsing systems.

1.1 Usability and security

Among Tor's users are civil and human rights activists in despotic regimes. For these people, Tor's anonymity is a critical security property that protects them from surveillance. [1]

There are three distinct levels at which Tor must to be usable in order to be effective:

1. Its usability must uphold the primary task, web browsing — a web browser should be usable for web browsing.

2. Its usability must uphold the secondary task, secure anonymity [WT99] — it should be hard to incorrectly use in a way that defeats the user's security.

3. It must have usability that encourages an increased user base — anonymity loves company [DM06].

It is the first and third of these points that this thesis seeks to improve. The Tor browser gives a relatively slow and therefore less usable web browsing experience, which is likely harming its uptake, and by extension the security provided.

[1]It is not an unfounded concern — today Martin Luther King Jr. is universally celebrated, but in his lifetime he was the target of surveillance [Com76]:

> From December 1963 until his death in 1968, Martin Luther King Jr. was the target of an intensive campaign by the Federal Bureau of Investigation to 'neutralize' him as an effective civil rights leader.

The NSA also surveilled King, under *Operation Minaret* — but later concluded the operation had been "disreputable if not outright illegal" [Pil13]

Chapter 2

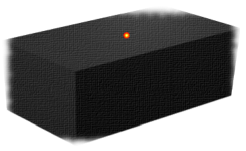
Background

Tor's network transport and flow control has changed little since its inception, and over time it has become clear that a new transport subsystem could yield improvements to Tor browser performance.

For better understanding, this chapter details some background technical information relating to transport and flow control in general and as implemented in Tor.

2.1 The Internet

The following assumptions of the internet are made throughout this thesis and in the experiments conducted:



- There are two commonly deployed versions of internet protocol, *IPv4* and *IPv6* — the more prevalent IPv4 [Pos81a] is assumed.
- The more prevalent physical layer *Ethernet V2* (without jumbo frames) is assumed, resulting in an IP maximum transmission unit (MTU) of 1500 bytes.
- All packets are assumed to fit within the MTU, so there is no need for IP fragmentation.

Recall that in IP, packets are transmitted by a sender and routed towards the receiver on a *best effort* basis, meaning that at any point the packet may be dropped with no notification to either party.

2.1.1 IPSec

IPSec is a protocol suite that allows for secure encapsulation of encrypted IP packets on top of IPSec IP packets [KS05]. It can be used as an encrypted tunnel for transporting IP packets between two hosts. It may also be used in a VPN arrangement, with an IPSec host forwarding a packet potentially through another IPSec tunnel.

IPSec is implemented in the Linux operating system kernel.

2.2 Transport Protocols

The major protocols used for transport over the internet are TCP [Pos81b] and UDP [Pos80].

2.2.1 UDP

For UDP, the data communicated is consisting of *datagrams.* These are individual messages containing 0–1500 bytes of data sent as a single IP packet. The protocol is described unreliable because there is no mechanism to notify either party or guarantee delivery of the message if the packet is dropped.

The minimalist semantics of UDP and its wide support on the internet make it a good substrate for other network protocols.

2.2.2 TCP

TCP by contrast is a connection-oriented protocol. Once a connection has been established between two parties, the connection may be viewed as a bidirectional data stream by an application.

The operating system (assumed throughout to be the Linux Kernel or simply *kernel*), is responsible for reliable delivery of data that is “sent” by an application using TCP.

TCP packets contain a *seq* number and an *ack* number. The seq number of a packet identifies the offset into the data stream (in that direction) of the data contained in the packet, while the ack number is used to communicate the seq number (in the other direction) that is needed next from the other host. The ack number implies that data for all previous bytes sent by the other host has been correctly received.

It is common for empty packets not containing any data to be sent, purely for the purpose of communicating a higher ack number to the other host. Additionally, it is common for hosts to re-send the same acknowledgement for each packet they receive with TCP data that is beyond the next byte expected.

In this way, a host can become aware of probable packet loss in a packet it has sent if: no acknowledgement arrives within a reasonable time; or, if multiple acknowledgements arrive containing the same earlier seq number [APB09].

TCP uses a sliding-window mechanism for flow-control. A sender may only transmit a window of data up to a certain size to the other host, and at that point must not send more data until the window size increases, or some of the earlier data is acknowledged. The receiving host specifies the window size in packets returned to the sender.

TCP implements congestion control [APB09] so that connections on the same link can make efficient use of available capacity without causing over-congestion or congestion collapse. This works firstly by congestion avoidance — hosts will enter a "slow-start" mode and build up transmission rates until loss is detected. Once loss is detected, a TCP host will greatly reduce the transmission rate, and return to slow growth from there. These transmission rates are implemented using a congestion window determined by the sender. At any time a TCP host may only send data if both the flow control window and the congestion window allow.

An aim of TCP congestion control is for competing streams to achieve *max-min* fairness. This is a state where no connection could use more bandwidth without another of the same or less bandwidth having their share reduced further. This round-about definition does allow for imbalances, so long as the imbalance is not to the detriment of any connection that has a lower share.

CUBIC [HRX08] is a widely deployed congestion control algorithm that is used in the experiments of this thesis. CUBIC increases the congestion window in three phases. At a congestion event, CUBIC reduces the congestion window by a factor. From here, absent a congestion event, it increases the congestion window rapidly up to the previous window size, then enters a steady state period, and then if a congestion event has still not occurred, will rapidly increase the congestion window until it finds a window size at which a congestion event does occur. At this point the congestion window is reduced and the process starts anew.

The term *RTT*, for *round-trip time* has special relevance in TCP, where it is calculated as a function of recently observed acknowledgment times for outbound packets, and is used to detect loss events.

The *bandwidth-delay product* of a link is its bandwidth rate multiplied by its RTT. The resulting value is the amount of data that must be in-flight at any time to fully utilise the link. Consequently, TCP window sizes should be at least this large, to allow enough buffer-time to acknowledge received data and move the flow control window forward.

2.2.3 TLS

TLS (currently Version 1.2 [DR08]) is the standard for cryptographically protecting the confidentiality, integrity and authenticity of data transferred over a TCP connection.

2.2.4 DTLS

DTLS (currently Version 1.2 [RM12]) is a variation on TLS designed for transportation over datagram protocols, particularly UDP.

2.2.5 SCTP

SCTP [Ste07] is a transport protocol that is quite similar to TCP in allowing a reliable congestion-controlled channel. An additional feature of SCTP is that data is tagged with a *stream identifier* and a *stream sequence number*. As a result, a lost packet for one stream need not prevent SCTP from delivering stream data destined for another stream that wasn't impacted by the packet loss.

Of note, SCTP streams do not have independent flow or congestion control. This means an SCTP connection behaves similarly to a single TCP connection in terms of max-min fairness, regardless of the number of streams transported.

It also means that a receiving application should not discontinue reading from any stream if it still has interest in reading from any another — due to the reliable semantics of SCTP it must retain data it has acknowledged, and eventually the unwanted stream buffer may grow full so that SCTP is unable to acknowledge any further data for the connection.

Since SCTP is not as widely deployed on the internet as TCP and UDP, it is often encapsulated using UDP packets [TS13] instead of plain IP packets.

2.2.6 QUIC

QUIC [HIS$^+$16] is similar to the combination of DTLS-SCTP-UDP.

In contrast to SCTP, streams in QUIC have independent flow and congestion in addition to QUIC's per-connection flow and congestion control.

A signature feature of QUIC is to allow for 0-RTT connection establishment, which is possible when the connection initiator has cached key information for the other host from a previous successful connection [LC16]. If so, the initiator will use *initial* encryption mode with that key information to send data, until it has received ephemeral key information from the server in response.

A non 0-RTT connection will send an *inchoate hello* to elicit the ephemeral key information in response. Once the initiator has received ephemeral key information, it moves into *forward secure* mode for the rest of the connection.

The QUIC protocol includes potential improvements over TCP such as NACK ranges [ISG15]. Other improvements being considered include packet pacing, speculative retransmission and Forward Error Correction of lost packets.

2.2.7 µTP

µTP is a UDP-based transport providing reliable data transmission similar to TCP. Of note, µTP uses a *delay-based* congestion control, that will tend to cede to other traffic such as TCP connections when other traffic appears to be using the link.

2.3 Tor

All discussions of Tor will in fact refer to the Tor version *0.2.7.6*. This is a relatively recent stable version of Tor that was used in the experiments conducted. A significant majority of Tor relays run on Linux, so for ease of discussion that operating system is assumed.

The Tor network consists of several thousand volunteer-operated relays. When a user installs the Tor client software on their computer, it selects a small number of these relays as semi-permanent *guard* relays, meaning those relays will be the first point of contact for the client's future connections over the Tor network.

The standard unit of transfer in Tor is called a *cell*, which is typically 512 or 514 bytes in size [DMS04] (a uniform length aids traffic analysis resistance). The unit of data transfer is the *relay data* cell, containing up to 498 bytes of application data.

In the background, the Tor client software constructs a small number of paths through the Tor network that are ready to be used when a user wishes to create a TCP connection through Tor. The paths are 3 hops long, with a guard relay as the first hop. The other relays, named *middle* and *exit* relays are selected at random, weighting towards higher bandwidth relays as published in a consensus file.

There is typically one TCP connection between each pair of relays that transfers cells with each other (accidental duplicate connections are disregarded).

A client establishes successive encryption layers with each of the relays, such that the guard relay may only decrypt the outermost part of a relay cell, the middle only the outermost of that, and finally, only the exit is able to decrypt the remainder. This arrangement is termed *onion-encryption* due to the successive layers of encryption.

The path as used for transport of the client's TCP connection is called a *circuit*. Tor network relays will typically have many active circuits at any one time, and many circuits may be using the same relay–relay pair at any time, all of their cells being transferred over the same TCP connection.

Tor cannot rely on TCP's flow control to regulate traffic on a circuit — as to do so would affect all other circuits using the same TCP connection too. Instead, clients and exit relays observe a *packaging window* limit in data they send to one another. The packaging window is decremented by one for each cell transmitted, and once it has reached zero the sender must stop transmitting data. On receipt of a certain number of cells, the receiving end returns a *SENDME* cell to the sender, which allows the sender to increase its packaging window.

2.3.1 libevent

Tor uses the *libevent* library to learn in an *operating system agnostic* way that an input/output event on a TCP connection has become possible.

2.3.2 Relay data path

A whistle-stop tour of a Tor relay's data queues is called for. A relatively complete picture is provided, with the intention of demystifying the process and describing the relevant components inside a larger map of which they are a part.

Figure 2.2 outlines the queues along Tor's network data transfer path for a typical Linux host. *rxqueue* receives packets from the network interface, and the kernel will then process the packets as fast as possible. If packets arrive too fast for this, they will simply be dropped by the network interface.

In processing, the kernel will place the packet data in the appropriate *Receive Buffer* for a TCP connection, which should typically be sized to match the bandwidth-delay product of the connection. By default Linux will auto-tune the buffer size as needed.

The application uses the *read* syscall or similar to remove data from the buffer. If it does this too slowly and the *receive buffer* becomes full, Linux will advertise a window size of 0 in TCP packets it sends back along that connection. When this happens, it is termed *backpressure*, because it causes the sender's *send buffer* to start accumulating data until itself becoming full, allowing the condition to propagate backwards further.

A connection *inbuf* is an unbounded data buffer inside Tor. Tor typically reads data into the inbuf as fast as possible, but will however stop reading from a connection when: the internal circuit queue for an edge connection reaches a

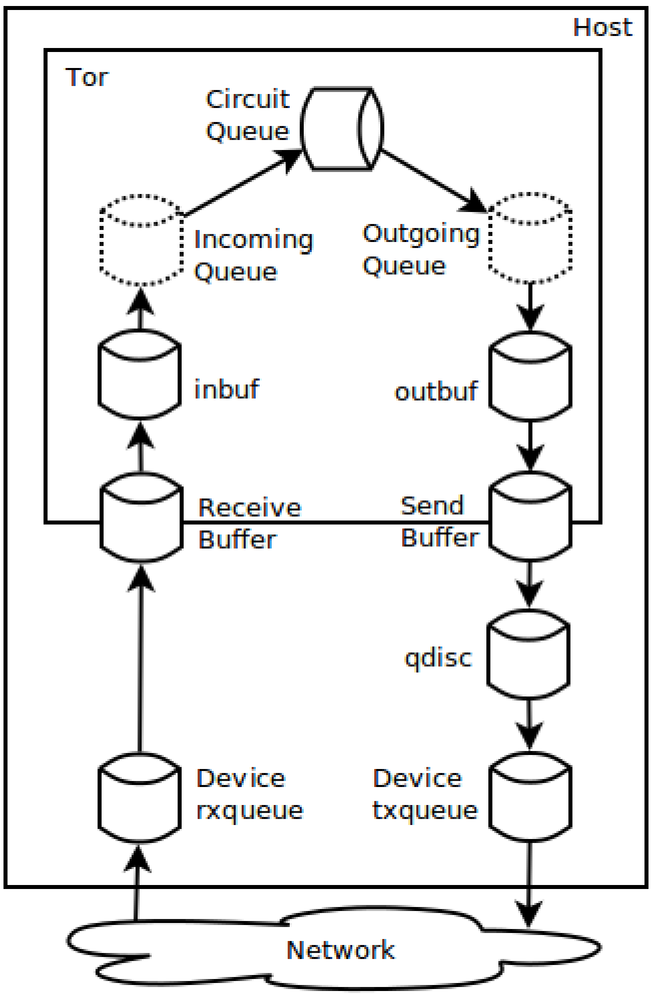


Figure 2.2: Queues in the Tor data path

certain size; when the packaging window for an edge stream using that connection reaches 0; and when the connection read rate limit or global read rate limit is exhausted. The data is immediately processed into cells by Tor and passed onwards.

Incoming Queue is typically not used, but its existence is noted here for completeness.

Aside from the case of an edge connection noted above, the internal *circuit queue* is unbounded in length, relying instead on Tor's packaging window to enforce a sensible upper bound.

Cells rest on the circuit queue until they can be scheduled for writing. The cells are written onwards from the circuit queue as fast as the EWMA scheduler [TG10] will allow for the circuit, provided the outgoing connection appears to be writeable and not blocked.

Outgoing Queue is typically not used, but its existence is noted here for completeness.

outbuf is a data buffer for serialised data ready to be written to the connection. Data rests here until processed by the libevent connection write callback. The scheduler will stop appending data to this buffer once it exceeds a certain length, leaving the cells on their circuit queues instead.

The libevent write callback will fire once its *Send Buffer* has space to accept more data. The data will be removed from outbuf and passed into the buffer using the *write* syscall or similar. If the send buffer becomes full, no further writes are allowed for the application on that socket until it is no longer full.

As permitted by flow and congestion control, the kernel will split the data into TCP segments and pass the resulting packets onto the *qdisc* queue. The qdisc queue is a space for *queueing discipline* algorithms to be implemented. Of relevance is the *netem* queueing discipline, which allows various network conditions to be emulated using the queue, such as varying latency delays, and increased probability of packet loss.

Lastly, packets are passed onto the device *txqueue* for transmission onto the network.

Chapter 3

Related work

Various alternative schemes have been proposed over the years to replace the single TCP connection per relay-pair that Tor currently uses for transport. It is helpful to group them in categories based on the reliability properties of their network design [Mur11]:

- Hop-by-hop — relays are only responsible for reliability of data in the network until it is acknowledged by the next hop. The next hop is then fully responsible for successfully sending the data onwards, and so on. Tor's transport currently follows this design.

- End-to-end — packets communicated to and from the client have a close correspondence to the IP packets sent and received by the the end server. As such, the client and server are responsible for retransmission of data if any IP packets or their equivalents are lost.

- Initiator-to-exit — similar to end-to-end reliability, but only up to the exit relay. From there, the exit relay is responsible for reliable transmission to and from the server. There may be some value in distinguishing the hosts running Tor software from those outside of the Tor network.

3.1 Initiator-to-exit

Viecco investigated a network design using unreliable transmission between the client and the exit relay, named *UDP-OR* [Vie08], with fairness being one of the main goals. A later work investigated this topic in detail [TS11] and appears to corroborate that *UDP-OR* is a fair approach, if not an easily deployable one. The *UDP-OR* design is not compatible with Tor's current design so deployment would be difficult in practice [RG09].

Regarding performance, more testing of the design would be need to be able to say how it fares in practice in comparison to Tor.

3.2 End-to-end

Kiraly et al. used a nested "telescope" of IPSec tunnels in place of Tor's TLS and onion encryption. They tested their design and achieved good results. They did not test their design under a high latency setting with additional packet loss however [Cla16b] (see below for why this is important).

3.3 Hop-by-hop

An obvious design alternative would be to modify Tor to open one TCP connection per circuit. There are two disadvantages to this [DM09]: Firstly, there is a reduced degree of mixing of cells, since it becomes clear that all cells on a given TCP connection are for the same circuit, as opposed to potentially any of the circuits sharing that hop.

Secondly, the larger number of connections increases the number of open sockets that must be maintained by the host, potentially harming scalability or allowing a socket exhaustion attack as later corroborated by Geddes, Jansen and Hopper [JGH14].

Motivated by TCP's head-of-line blocking, Reardon and Goldberg [RG09] implemented an alternative transport named *TCP-over-DTLS*. Using a user-space TCP stack, pseudo-TCP packets are transported between relays over UDP with

DTLS. This gave positive initial results, though a suitable TCP stack for live deployment remains elusive.

AlSabah and Goldberg implemented an analogous approach named *PCTCP* [AG13] with an IPSec tunnel in place of DTLS over UDP, allowing use of the host TCP stack. They extensively tested the design and showed significant performance improvements for web clients in large scale simulations. Under normal operation AlSabah and Goldberg expect PCTCP relays to use less than 10,000 file descriptors, but note the possibility of a scheme to fall back to multiplexing if such a high number is reached.

Geddes, Jansen and Hopper [JGH14] showed that how an adversary might cause aforementioned socket exhaustion in PCTCP, and proposed a scheme named *IMUX* to dynamically scale the number of connections. Their large scale simulation did not show a significant performance improvement of PCTCP over vanilla Tor, and showed a slight improvement for web clients under IMUX.

However, although solving the socket exhaustion problem, its use of normal TLS connections retains the previously voiced concerns over reduced cell mixing [DM09], and this issue is not addressed by the work. Further discussion would be needed to consider if this is an acceptable design trade-off.

Jansen et al. [JGW$^+$14] investigated delays of data in the Tor network, and found significant amount of time being spent in kernel buffers. This is unsatisfactory because once data leaves control of the Tor application, Tor can no longer schedule the traffic for best fairness and efficiency. They modified Tor to give it an awareness of the likely levels of congestion in kernel buffers, allowing data to remain in Tor until the last moment so it can be scheduled more optimally.

It was found that not only does Tor with KIST outperform Tor without KIST for web traffic [JGW$^+$14], but in fact vanilla Tor outperforms both PCTCP and IMUX for web traffic, when KIST is in use [JGH14].

3.4 Conclusions

Despite nearly a decade of academic work showing performance improvements, Tor's transport has changed very little. The 2009 report *Performance Improvements on Tor* [DM09], points to a number of possible reasons why this is the case: integration and deployment effort is high, often the components are not mature enough yet, and the solutions are complex, leading to a high risk of issues in real-world use. It is therefore advisable to hold these factors in mind while developing new proposals for Tor's transport.

3.4.1 Network design

As Reardon and Goldberg noted in concluding remarks, approaches other than hop-by-hop will incur an extra cost for retransmissions, since these must be rerouted through a larger part of the network [RG09].

As Tschorsch and Scheuermann discuss [TS12], due to the longer RTT of TCP connections, end-to-end approaches will also take longer to "ramp up" through slow start and up to a steady state.

Both of these factors (not to mention increased security risk of information leakage [DM09]) suggest that hop-by-hop designs are likely to yield better results. In fact, the hop-by-hop approach may be viewed as an instance of the *Split TCP* Performance-Enhancing Proxy design, whereby arbitrary TCP connections are split in two to negate the issues noted above.

3.4.2 Transport protocols

The load balancing algorithm of the Tor network is reasonably straightforward — each client selects middle and exit relays randomly, weighting to historical bandwidth statistics. This means that a relay which fluctuates significantly greatly in bandwidth is likely to be either over or under-subscribed at times.

Circuits should therefore be competitive in claiming their bandwidth, so that fluctuations in background traffic have a reduced impact. Absent from discussion [Cla16b] by Tschorsch and Scheuermann [TS12], Tor allows a global limit to be set on its bandwidth usage. In the context of Tor's load balancing algorithm, it is therefore better for an operator to set a low bandwidth limit that Tor will compete effectively for, than to set a high limit that Tor might cede almost entirely to other applications using the same link.

As such, µTP, SCTP streams, and a single-TCP connection as currently used by Tor should be considered less favourable choices for transport. µTP will cede to competing traffic by design, and the latter two will compete with other traffic at the connection level instead of the circuit level, making them more prone to cede to other traffic.

Remaining good choices are multiple TCP connections over an encrypted tunnel (such as TCP-over-DTLS and PCTCP) and QUIC streams. Single-stream SCTP connections may also suffice as an alternative to TCP connections. Another potential advantage of these schemes is that their independent congestion control could in future allow backpressure along individual circuits, without affecting other circuits on the same tunnel.

While there are pros and cons of the approaches, user-space-transport-over-encrypted-UDP options are attractive compared to IPSec-based approaches for the following reasons:

- Deployment — unlike in-kernel IPSec, no administrator privileges are required to install or run user-space software [RG09].
- Sockets — user-space transport implementations don't require sockets for each connection, minimising the potential for socket exhaustion [DM09].

The maturity of user-space TCP stacks still appears to be a limiting factor [Mur11], since these stacks are typically developed for research purposes and do not appear to have not gained significant traction (and consequently support) in industry.

SCTP and QUIC appear to be mature options, with SCTP being widely deployed in Firefox and QUIC widely deployed in Chromium.

Tunneling single-stream SCTP connections over DTLS seems a more unusual design choice than to use QUIC directly, and as a more recently designed protocol, the QUIC protocol includes improvements such as NACK ranges. QUIC's 0-RTT (re)connection ability between hosts doesn't figure strongly in decision-making, since relay–relay connections in Tor tend to be very long-lived, and so their setup latency is less likely to have a major impact on performance.

In conclusion, after review of transport technologies and related work, a hop-by-hop QUIC transport appears to be a promising alternative transport design for Tor. It is better able to utilize Tor's global bandwidth limits, or the link capacity otherwise, doesn't suffer from head-of-line blocking, and could allow for per-circuit backpressure in future. Additionally, its user-space implementation allows Tor to make use of its advanced transport protocol features regardless of the operating system kernel version it is running on [RG09].

While there is no existing peer reviewed literature evaluating use of this approach, it is not a new idea — two separate groups [KL16] [AEOAE16] have researched this approach in parallel, and had begun to do so before this thesis topic was chosen.

Nonetheless, this thesis uses a distinct bottom-up reasoning process for its motivation. As a result, QUIC's 0-RTT connection ability doesn't constitute part of the motivation for this proposal. The use of 0-RTT initial encryption mode brings a potentially complex design trade-off between performance and security.

Additionally, backpressure is explicitly noted as part of motivation for this proposal, as contrasted with the lack of backpressure capability in SCTP's streams, since fairness [TS11] and flow control [DM09] [TS12] are still open problems in the Tor literature. The QUIC proposal, in conjunction with Tor's existing bandwidth limiting mechanisms, allows a straightforward design solution to this problem.

Chapter 4

Methodology

4.1 QUIC

A ground-up software implementation of the QUIC protocol would have been prohibitively expensive. The de-facto public implementation of the QUIC protocol exists in the Chromium web browser source tree. Since this isn't convenient for developers wishing to use QUIC, third party developers have created a fork of the source code named *libquic*, allowing the QUIC implementation to be built as a standalone library and linked into other programs. More recently, the QUIC developers have started maintaining an unofficial fork of Chromium's QUIC source code, named *proto-quic*.

libquic has been modified to minimise its code footprint and dependencies to the essentials, which made it a convenient starting point for building on, since this reduces the potential for complications in dependencies. However it is expected that with a small amount of additional work, proto-quic could be used instead with similar results.

Since the Chromium QUIC implementation is written in the same language as the Chromium browser, C++, only C++ programs may link with and use libquic. This is problematic since Tor is written in the C language. To work around this, a C++ library was needed to expose QUIC functionality through a C API.

The only library known that provided this functionality was a library named *quicsock* hosted in a repository on *github.com* named *simple-quic* [Li16], created by students investigating the same topic. However, the library is not freely available, having since been removed from github. Additionally it has a multithreaded design that could result in a false negative performance result, and would likely not scale to the size of the live Tor network.

A C++ library (unoriginally) named *libquux* was therefore developed to provide a C API to libquic's functionality.

4.2 QUUX

The libquux library is designed to be used asynchronously by allowing callbacks to be set, which the library then "calls back" in an edge-triggered fashion when a condition is met, such as libquux being able to write more data to a stream.

There was insufficient time to implement certificate verification, since this would require integration with Tor's verification mechanisms, however due to the long-lived nature of most relay-to-relay connections this is unlikely to affect applicability of test results to Tor.

An array of coding techniques was utilised to reduce the possibility of a false negative due to the experiment code, including: use of the *mmsg* family of IO functions, caching of time values, whole-program compilation, and constness. Where found to have been premature optimisations, these can be reverted in future. There are likely to be cases where a faster implementation was used than could be used in a real deployment. It is believed there aren't any cases where this would have had a significant impact to the experimental results, if at all.

Notwithstanding the last hurdle of certificate verification, *libquux* [Cla16a] is the only freely available and open source QUIC C API library known to exist at the time of writing. An abridged header file for libquux is attached in appendix A.1.

4.2.1 Linking

Having developed a QUIC library with functions of C linkage, the library was linked into the Tor executable along with libquic, statically built dependencies libprotobuf and BoringSSL's libcrypto, and the stdc++ dynamic runtime library.

BoringSSL is a fork of OpenSSL used by the Chromium web browser, however it still uses the same symbol names as OpenSSL. Since Tor already has OpenSSL as a dependency, linking both libraries into an executable would result either in a failure due to duplicate symbols, or in only one of the libraries being used for its version of the duplicate symbol throughout the program, potentially leading to a change in semantics.

To avoid this problem, a small program was created to rename all of BoringSSL's symbols to have a *bssl_* prefix, and libquic was compiled with original symbol references automatically replaced with their *bssl_* variant.

4.2.2 Channels

In previous experiments, Google found that QUIC connectivity would fail for approximately 5-10% of users [Ros12]. In those situations the Chromium browser will silently fall back to using TCP. It's likely the connectivity of Tor relays hosted in data centres is better, however similar behaviour would likely be desirable at least for Tor clients.

While it should be possible to develop a variant of Tor that creates a QUIC connection in parallel with a TCP connection as Chromium does, the change would likely be complex and may require duplication of some of Tor's protocol handshake logic.

An alternative approach used previously in a similar experiment [LMJ13] is to allow the Tor handshake to complete over TCP and only then attempt to use parallel connections. This method requires smaller and more orthogonal changes, with most of the code being additions rather than changes to the existing code. Additionally, it would be relatively straightforward to use QUIC for the channel only if it successfully connected before any cells were ready to be sent, and otherwise continue using TCP for the remainder of the connection.

Once the Tor handshake is complete, the connection initiator starts sending the cell(s) for the circuit which caused a connection to be created in the first place. A straightforward implementation needn't hook into any circuit creation logic for this; instead when asked to send a cell for a circuit that hasn't previously been used, a new QUIC stream is created, and the cell immediately sent on that stream.

Since Tor currently doesn't have a concept of circuits having their own connection, there isn't a mechanism to block writes for one circuit on a channel without blocking other circuits on the channel. This is relevant to the experiment at hand, because channel-level blocking could re-introduce head-of-line blocking internally. It proved too difficult to apply per-circuit blocking inside of Tor; instead writes on the channel were made to always succeed by placing data into per-circuit buffers if the corresponding stream had become blocked. As a compromise, a branch of Tor named *0.2.7.6-patched* was modified with a similar change, to verify that this change alone was not the cause of any performance differences.

4.3 Chutney experiment

An experiment was designed to test whether QUUX showed an improvement as predicted by head-of-line theory. The experiment involved two hosts either side of the Atlantic to ensure natural latency in connection paths between the two. As outlined in figure 4.1, the experiment consisted of a minimal Tor path with the Guard and Middle being on one host, and traffic source, a varying number of Tor clients (1–8), exit relay and traffic destination being on the other. A varying amount of loss (in exponentially scaled quantities) was emulated for the middle–exit link using *netem* on the *eth0* device.

In each scenario, all clients sent 32MiB to the verify server over the test network in parallel with each other. The time taken for the slowest client to finish sending was recorded, to find the rate of its goodput.

Test runs were performed in the order of: tor-0.2.7.6, QUUX, tor-0.2.7.6-patched for all test variations. The proximity of test time ensured that each branch was likely to experience similar network and host conditions to the others. While some experiments may have had slightly favourable network and host conditions to one branch over the other, due the very large number of experiments conducted this effect is expected to cancel out in aggregate.

For each experiment run the experiment.2.sh (appendix B.1) script calls *configure-hosts.sh*, which completely resets the host states by killing any Tor processes,

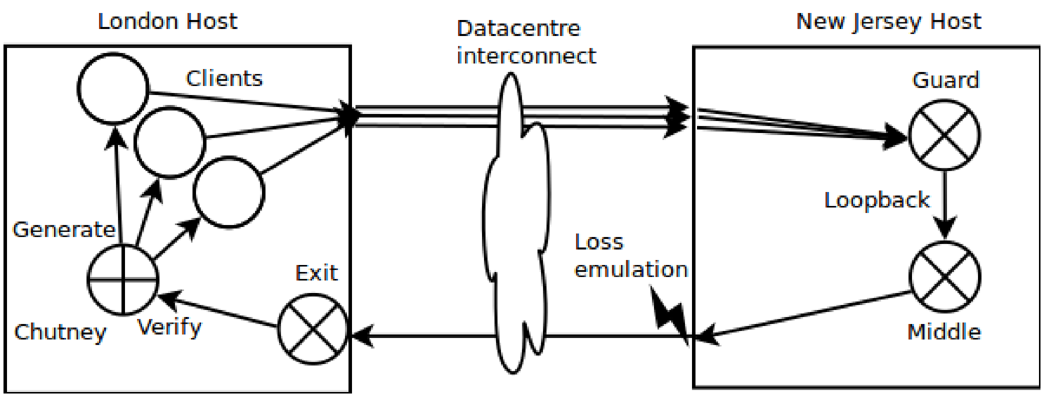


Figure 4.1: Overview of the Chutney experiment

removing any loss emulation still in place, and re-configuring the Chutney network for the next experiment being tested.

The London host had 8 CPUs and the New Jersey host had 4 CPUs, so with the exception of the 7 and 8 client tests, there was at least one CPU available for each process on the system.

Each line in figure 4.1 corresponds to a TCP or QUIC connection, with arrows pointing in the direction of data flow. In the case of a QUIC connection, there is one QUIC stream for each client circuit carried along the connection.

CUBIC was used as the TCP congestion control algorithm for the TCP stack of both hosts, which matches the algorithm used by QUIC.

Chutney allows a CHUTNEY_CONNECTIONS variable to be set, however to be sure that individual circuits and therefore independent QUIC streams were created, separate client processes were used instead.

It was found that each host used TCP Segmentation Offload and therefore allowed creation of much larger segments than the MTU at the kernel level. Since this doesn't occur for UDP, segmentation offloading was disabled on the loss-varying host to ensure a fair comparison with similarly sized packets being dropped.

Logging can have a large impact on the performance of applications. Since the QUUX and Tor branches exercise different code paths with potentially different amount of logs generated, logging was disabled entirely during test runs for a fairer comparison.

4.4 Shadow experiment

The tor-0.2.7.6, tor-0.2.7.6-patched, and QUUX branches were tested under the built-in Shadow config *shadow-toy-config*. This configuration contains 100 web servers, 3 authority relays, 66 middle relays, 14 relays flagged Guard, 10 relays flagged Exit, 5 relays flagged Guard and Exit, 360 web clients, and 40 bulk-download clients.

Web clients fetched 320KiB files from the web servers at dispersed intervals, whilst bulk clients downloaded 5MiB files as background traffic in the network. The first 30 minutes of the 60 minute simulation are ignored, since during this period the network is still bootstrapping itself.

Although this experiment doesn't simulate an accurate representation of the live Tor network, it should be sufficient to observe relative performance differences between branches. Default provision of the test configuration with the Shadow software also reduces the barrier for comparison with other branches in future.

It's important for scientific experiments to be repeatable. The Shadow experiment was conducted from within a Docker container, and the Dockerfile for setting up the QUUX experiment is provided in appendix C.1, making repeatability much easier for the experiment. About 35GiB of memory is needed to run the experiment for the full 60 minutes, however 24GiB is enough memory to run the experiment for about 50 minutes, which should be sufficient to achieve some initial results.

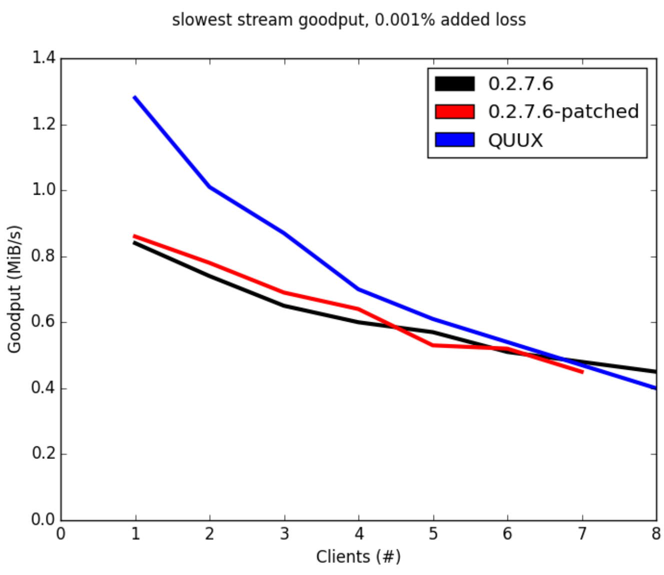
Chapter 5

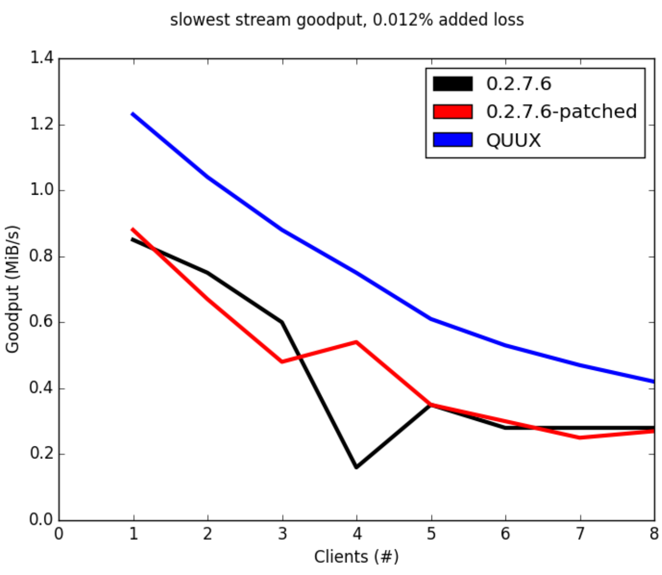
Results

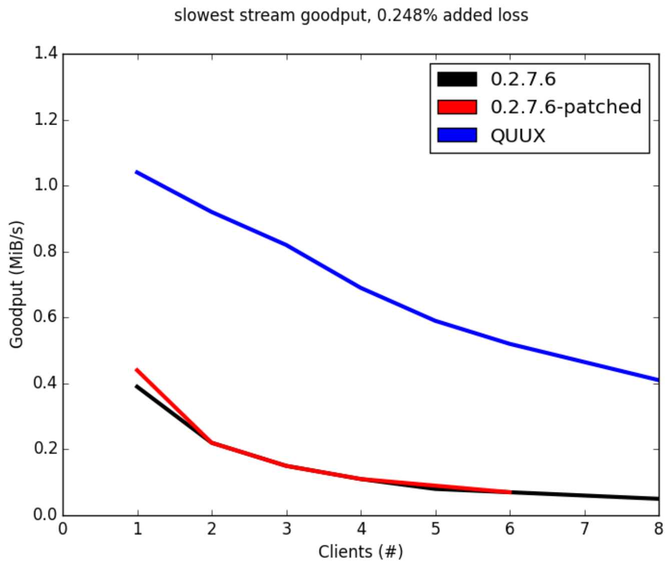
5.1 Chutney experiment

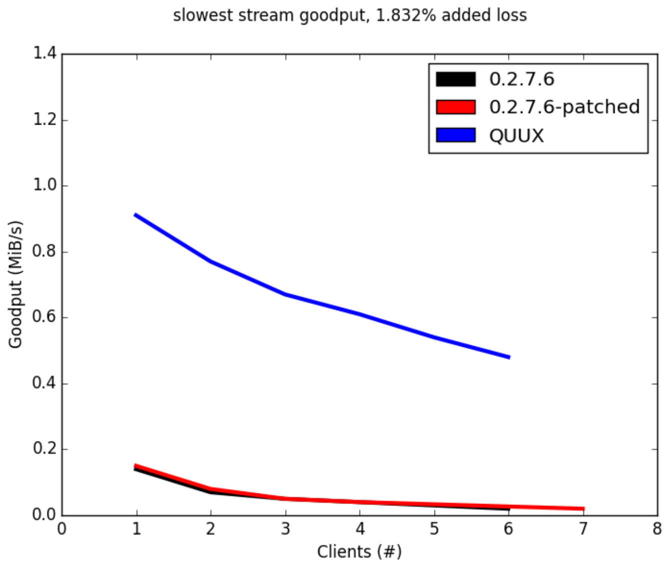
The following graphs show how goodput varies by the number of clients at different levels of background loss, for each branch.

Due to the large number of tests and traffic transferred in each test, each scenario was only tested once. Therefore exact position of individual points should be taken with a slight pinch of salt, however when taken together the points support each other within a clear trend.









5.1.1 Bandwidth overhead

Bandwidth overhead was measured on the London host for all traffic emitted by the Chutney process, the clients, and the exit relay.

The largest overhead seen was for egress bandwidth for 8 clients at 0.001% loss, which resulted in up to 25% bandwidth overhead. The smallest egress bandwidth overhead observed was 5.72%, which occurred for 2 clients at 1.832% added loss.

Unfortunately, since the data value was taken from an aggregate counter in the kernel using the *nstat* command, it also includes non-relay traffic to and from the Chutney process, which has now been conservatively (in tor-0.2.7.6's favour) subtracted.

In future, a better approach would be to direct client eth0 traffic and/or exit eth0 traffic through an empty iptables chain, allowing traffic counts to be queried using *iptables -L -v*, and reset after each experiment run.

5.1.2 Conclusions

The Chutney goodput results seem unbelievably good, but the experiment has been checked and re-checked — UDP packets are being dropped at the same rate, and QUUX makes no effort to learn about local send failures. Tor's highly averse reaction to packet loss — and QUUX's relative resilience — is reproducible using a straightforward *basic-min* Chutney test on a single localhost.

From the graphs looking at varying clients and 0.001% loss, we can see that QUUX goodput is much more a function of the number of clients than for Tor. Both branches showed a fairly straightforward nearly linear relationship between the number of clients and throughput at all loss levels.

Separately, the results do not show a significant client-specific performance degradation of multiplexing under a shared lossy link for Tor, and therefore QUUX did not show an improvement in relation to loss with multiple clients. This result is contrary to the theory that head-of-line blocking is a significant performance issue for Tor.

A possible explanation for this surprising result may be that the middle–exit connection was not the bottleneck in the experiment. In that case the head-of-line issue may indeed be a cause for slowness on that link, but the slowness is being masked by an even slower section of the path.

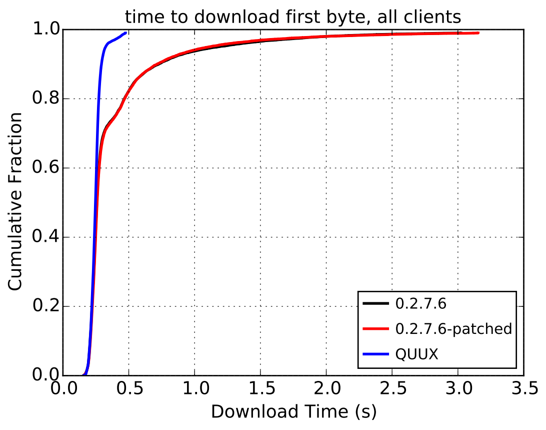
Since netem would also drop a percentage of the ack packets returned from the New Jersey host for client-to-guard traffic, this might be slowing the client–guard connection until it becomes the bottleneck.

Otherwise, it may be that compared to the performance impacts from congestion control, the head-of-line cost is not as great by comparison. A good next-step for investigation would be to compare Tor's performance when using one TCP connection per circuit. Such a design may not be desirable for live deployment, but might be helpful for understanding the performance impact of head-of-line blocking.

The fact that there is such a marked improvement at 0% packet loss for a single client is indicative of either an integration improvement in the Tor application, or a network protocol improvement.

That QUUX performance degrades significantly more slowly under loss than Tor indicates that QUIC protocol improvements are the likely source of performance gain. A hypothesis is that its improved ability to signal loss with NACK ranges is the source of credit, however further investigation would be needed to determine if this is the case.

5.2 Shadow experiment



The most important of the Shadow results are ones relating to small downloads, since they are indicative of web traffic performance. The 320KiB size approximately relates to the average web page size from a study in 2010 [JBHD12]. The 5MiB background traffic is intended to represent large file transfers such as BitTorrent traffic across the network, and as such its performance is not our primary concern when considering anonymous web browsing performance.

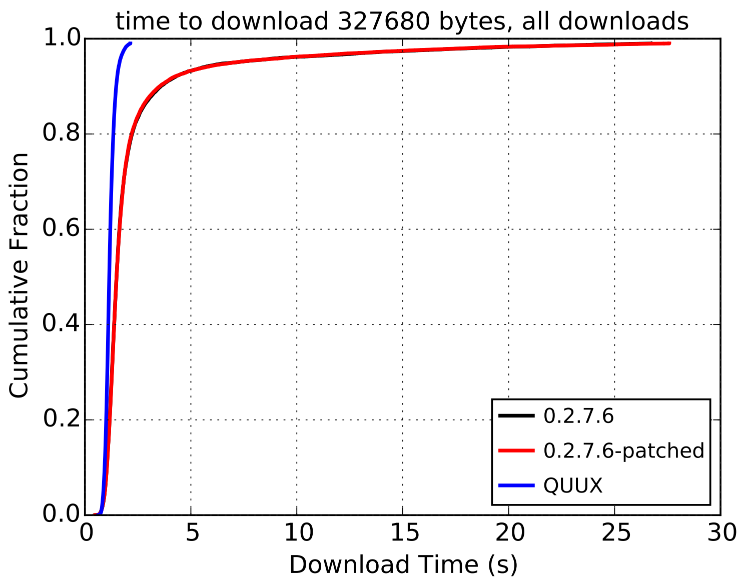
In practice, loading a web page is not as simple as downloading one 320KiB file — a smaller HTML file is first downloaded, and will typically reference a large number of resources from potentially many other web hosts.

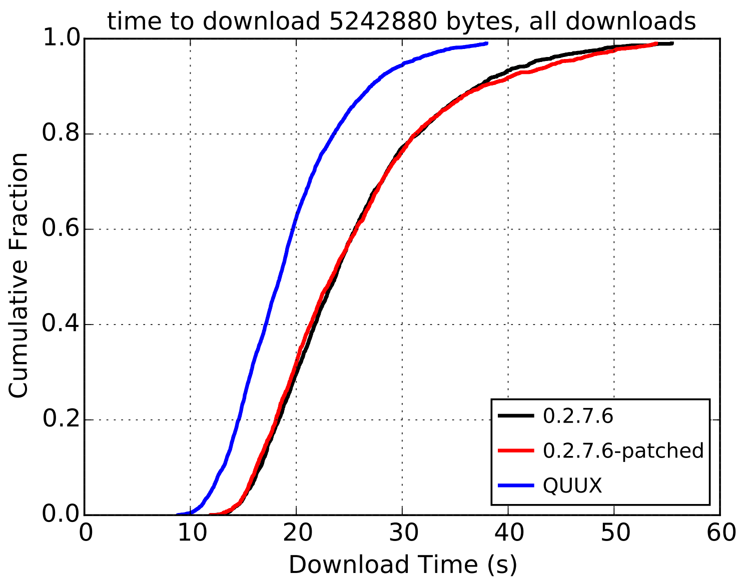
The page wouldn't be recognisable as a usable web page until at least the most important style resources have been downloaded. At that point however, the page may be largely usable, regardless of how many outstanding resources may still be loading.

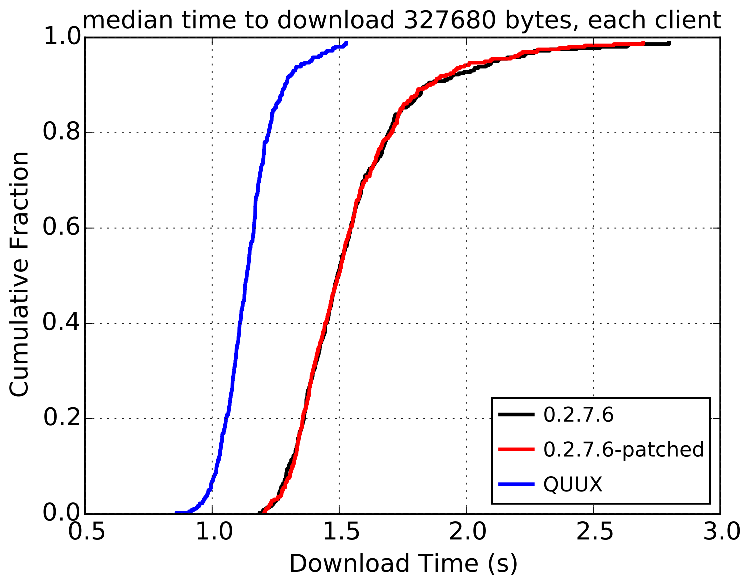
All of this is to say that the time-to-first-byte metric is very important in its own right, since it represents the minimum amount of time before new resources such as style-sheets can start to be downloaded. Experimentally [JH12], the 320KiB download format has also been found to result in a representative model of real network performance.

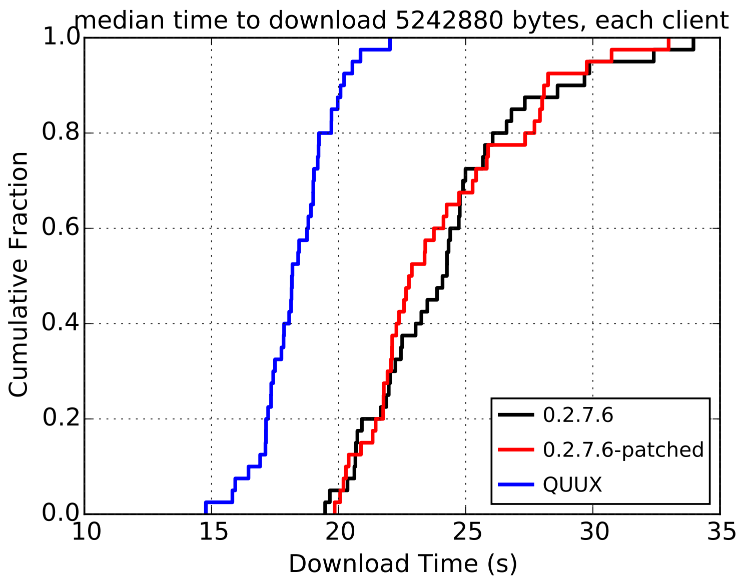
The next-most important metrics are the time to download 320KiB for all downloads, and the median time to download 320KiB for each client, as these represent typical web browsing performance.

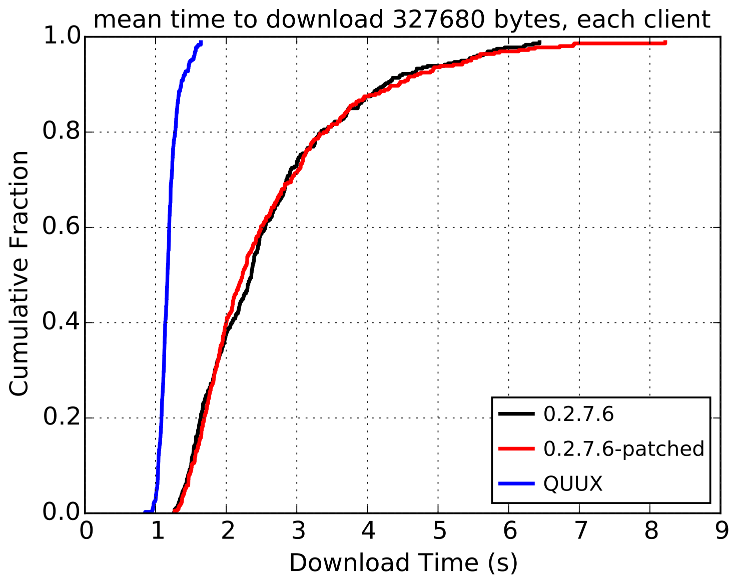
The max time to download 320KiB (and by extension, the mean) may not be as important to performance as it first appears, because at a certain point, users have the option to restart the connection, albeit a very annoying task.

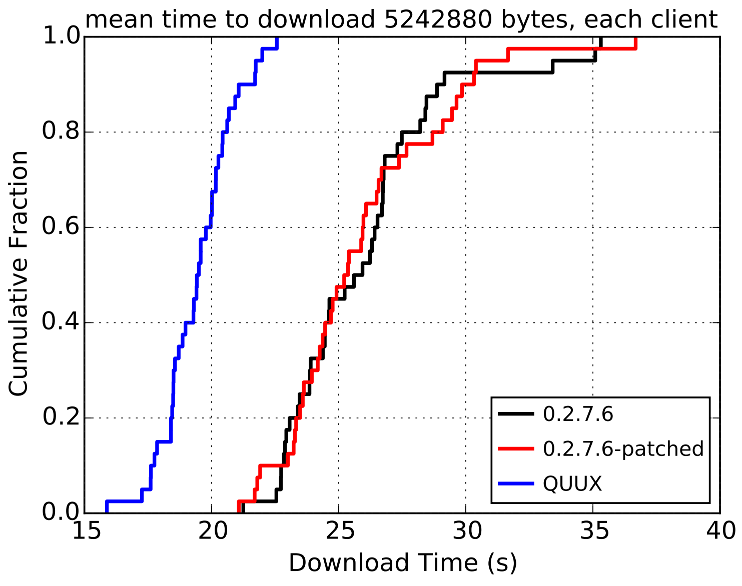

time to download 327680 bytes, all downloads
Cumulative Fraction
Download Time (s)
0.2.7.6
0.2.7.6-patched
QUUX

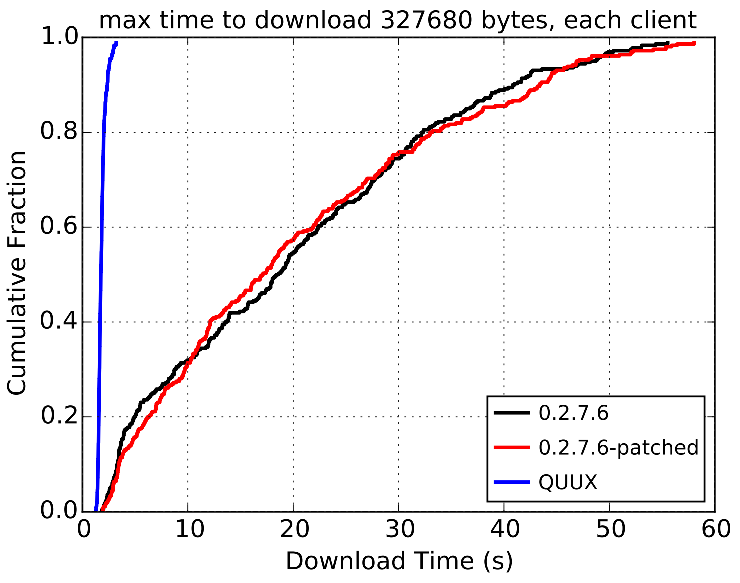

time to download 5242880 bytes, all downloads
Cumulative Fraction
Download Time (s)
0.2.7.6
0.2.7.6-patched
QUUX

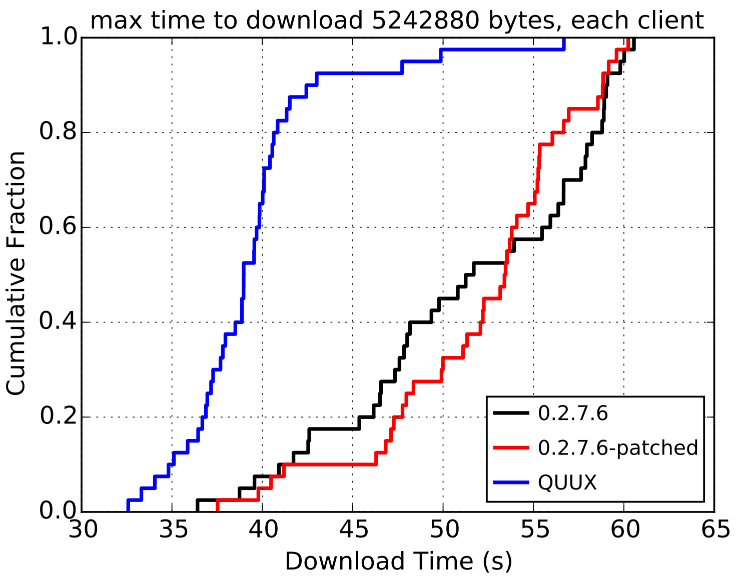

median time to download 327680 bytes, each client
Cumulative Fraction
Download Time (s)
0.2.7.6
0.2.7.6-patched
QUUX


median time to download 5242880 bytes, each client
Cumulative Fraction
Download Time (s)
0.2.7.6
0.2.7.6-patched
QUUX


mean time to download 327680 bytes, each client
Cumulative Fraction
Download Time (s)
0.2.7.6
0.2.7.6-patched
QUUX


mean time to download 5242880 bytes, each client
Cumulative Fraction
Download Time (s)
0.2.7.6
0.2.7.6-patched
QUUX





5.2.1 Conclusions

The results are unequivocal: The Tor network is substantially faster under QUUX, both in terms of throughput and time-to-first-byte.

From the time to download 320KiB for all downloads, we can see that for the slowest 25% of downloads, QUUX starts showing significant improvements over Tor. Particularly for the slowest 5% of downloads, QUUX retains reasonable performance while Tor performance becomes unacceptably slow or would need the request to be restarted.

QUUX shows approximately 20% improvement for the median download of the median client. The graph of median 320KiB download times for each client is indicative that all clients can expect to see performance improvements for typical web requests using QUUX, and not just the fastest or slowest.

To state these results to any greater precision would be misleading of the accuracy of simulation compared to real-world Tor performance. Instead, an effort has been made to make QUUX easily testable in future experiments.

Chapter 6

Conclusion

6.1 Backpressure

A key benefit of the QUIC protocol for Tor is to make circuit-level backpressure possible.

A fully QUIC relay path (with slight modification to fix a limit on internal buffer sizes) would allow end-to-end backpressure to be used from the client application TCP stream up to the exit TCP stream. Leaving aside Tor's inbound rate limit mechanism but retaining the global outbound limit [1], this design would allow max-min fairness to be achieved in the network, as outlined by Tschorsch and Scheuermann [TS11].

A last hurdle remains, which is that as with any whole-network change, migration path anonymity set concerns [Mur11] would apply to this change. Having upgraded a sufficiently large amount of the core network, consensus could switch so that only backpressure-enabled relays be accepted in circuits. There seems to be a graceful fallback for Tor clients that remain on the older versions — since a Tor client typically multiplexes traffic for a single user, it seems a reasonable upgrade path to apply backpressure on the whole client–guard TCP connection for clients still on older versions.

It remains for discussion if it would be reasonable for guards to be able to identify their older clients in this way. One option might be for newer clients to

[1]The rate of incoming traffic would follow from this, due to backpressure on the read buffer.

probabilistically use old TCP connections to interfere with a potential analysis.

Once implemented however, backpressure would allow Tor to adopt a significantly improved internal design. In such a design, a Tor relay could read a single cell from one QUIC stream's read buffer, onion crypt it, and immediately place it onto the write buffer of the next stream in the circuit. This process would be able to operate at the granularity of a single cell because the read and write operations for QUIC are very cheap user-space function calls and not syscalls as for host TCP [2].

The schedule of this action would be governed by the existing EWMA scheduler for circuits that have both a readable stream and a writeable stream (and as allowed by a global outgoing token bucket), allowing optimal quality of service for circuits.

It's expected that backpressure implemented in this way will yield significant performance and fairness gains on top of the performance improvement found in this thesis.

6.2 Security

6.2.1 Software

Software changes inevitably carry with them a risk of bugs, some of which have the potential to reduce security. Particularly relevant to Tor are any issues that could (partially) de-anonymise users, that risk denial of service in the network, or both.

The software change for the QUUX branch and libquux was relatively small — about one thousand lines of code in Tor and four thousand lines of code in libquux. Each of these changes would need to be thoroughly reviewed, and perhaps partially re-implemented for improved code clarity.

One of the motivations for using QUIC in this project was to depend on a mature software implementation already in use by millions of people in the Chromium web browser. Although much of the code is indeed reused, the client-side Chromium code uses its own subclasses of some core classes that were not

[2]Related to this, the location of buffers in user-space also allows for cheap KIST-style analysis

re-usable by libquux. Therefore careful review of the equivalent subclasses would be needed, to ensure libquic is used correctly in all cases.

An even larger difference between QUIC as used by Chromium-Google exists on the server-side. Since the Google server code is not open source, libquux must build on server code that was open sourced for demonstration purposes. This software would need an even more thorough review to ensure it is correct.

Finally, consideration would be needed towards trust of the dependency projects. To what extent can patches from upstream be trusted? Should Tor stick to a relatively static QUIC version, performing very costly code review and update infrequently? These are some questions that would need to be considered in adopting this proposal.

6.2.2 Usability

At the start of this thesis an assertion was made: the relative slowness of the Tor browser is likely harming its uptake. To what extent is this actually true? Results from e-commerce [For09] suggest an almost linear relationship between user engagement and latency that becomes measurable from as little as 100ms.

It may be the case that Tor users are less sensitive to latency however, understanding or even valuing it as a side-effect of the privacy technology [Mur07]. Further experimental work might be helpful to understand the impact of latency in the privacy setting on both user adoption and the primary task.

Dingledine and Murdoch formally analysed this relationship between performance and the number of users in the Tor network using supply and demand theory from Economics [DM09], building on an earlier analysis by Murdoch [Mur07].

Dingledine and Murdoch's point to a trade-off that shows relevance to this thesis, which is the trade-off between number of users and performance of web browsing. When translated to the derived security benefit from each, there appears to be a trade-off between the security of Tor as a usable privacy tool, and security of Tor as a web browser used by a large anonymity set.

But this needn't be the case: there *is* another option, which is to keep shifting the supply-side of Tor web browsing performance so it becomes highly usable for any number of people who wish to use it, and stays highly usable for any number of people who wish to use it. It's an ambitious goal, but not impossible.

Bibliography

[AEOAE16] Raik Aissaoui, Ochirkhand Erdene-Ochir, Mashael AlSabah, and Aiman Erbad. Poster: Qutor: Quic-based transport architecture for anonymous communication overlay networks. March 2016. https://www.researchgate.net/profile/Raik_Aissaoui/publication/292392094_QUTor_QUIC-based_Transport_Architecture_for_Anonymous_Communication_Overlay_Networks/links/56ae170008ae43a3980e6890.pdf.

[AG13] Mashael AlSabah and Ian Goldberg. PCTCP: Per-Circuit TCP-over-IPsec Transport for Anonymous Communication Overlay Networks. In *Proceedings of the 20th ACM conference on Computer and Communications Security (CCS 2013)*, November 2013.

[APB09] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), September 2009.

[Cla16a] Ali Clark. The libquux branch of libquic, 2016. https://github.com/aliclark/libquic.

[Cla16b] Ali Clark. Tor network performance — transport and flow control. Technical report, University College London, April 2016.

[Com76] Church Committe. Dr. martin luther king jr., case study. In *Book III: Supplementary Detailed Staff Reports on Intelligence Activities and the Rights of Americans*, page 81. April 1976.

[DM06] Roger Dingledine and Nick Mathewson. Anonymity loves company: Usability and the network effect. In Ross Anderson, editor, *Proceedings of the Fifth Workshop on the Economics of Information Security (WEIS 2006)*, June 2006.

[DM09] Roger Dingeldine and Steven J. Murdoch. Performance improvements on tor or, why tor is slow and what we're going to do about it. Technical report, The Tor Project, March 2009.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919.

[For09] Brady Forrest. Bing and google agree: Slow pages lose users, June 2009. `http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html`.

[Gor13] Siobhan Gorman. Nsa officers spy on love interests, August 2013. `http://blogs.wsj.com/washwire/2013/08/23/nsa-officers-sometimes-spy-on-love-interests/`.

[HIS+16] R. Hamilton, J. Iyengar, I. Swett, A. Wilk, and Google. Quic: A udp-based secure and reliable transport for http/2, 2016. `https://tools.ietf.org/html/draft-tsvwg-quic-protocol-02`.

[HRX08] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, July 2008.

[ISG15] J. Iyengar, I. Swett, and Google. Quic loss recovery and congestion control, 2015. `https://tools.ietf.org/html/draft-tsvwg-quic-loss-recovery-01`.

[JBHD12] Rob Jansen, Kevin S. 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*, Berkeley, CA, August 2012. USENIX.

[JGH14] Rob Jansen John Geddes and Nicholas Hopper. Tor IMUX: Managing connections from two to infinity, and beyond. In *Proceedings of the 12th Workshop on Privacy in the Electronic Society (WPES)*, November 2014.

[JGW$^+$14] Rob Jansen, John Geddes, Chris Wacek, Micah Sherr, and Paul Syverson. Never been kist: Tor's congestion management blossoms with kernel-informed socket transport. In *Proceedings of 23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA, August 2014. USENIX Association.

[JH12] Rob Jansen and Nicholas Hopper. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Proceedings of the Network and Distributed System Security Symposium - NDSS'12*. Internet Society, February 2012.

[KL16] Kevin Ku and Xiaofan Li. 15744 final project proposal: Improving tor performance with google's quic, February 2016. `http://archives.seul.org/tor/dev/Feb-2016/pdfLhhNT1x1Nw.pdf`.

[KS05] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005. Updated by RFCs 6040, 7619.

[LC16] Adam Langley and Wan-Teh Chang. Quic crypto. Technical report, Google, May 2016. `https://docs.google.`

com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit.

[Li16] Xiaofan Li. [tor+quic] progress update. email to tor-dev mailing list, April 2016. http://archives.seul.org/tor/dev/Apr-2016/msg00042.html.

[LMJ13] Karsten Loesing, Steven J. Murdoch, and Rob Jansen. Evaluation of a libutp-based tor datagram implementation. Technical report, The Tor Project, October 2013.

[Met] Tor Metrics. Direct users by country. https://metrics.torproject.org/userstats-relay-country.html.

[Mur07] Steven J. Murdoch. Economics of tor performance, July 2007. https://www.lightbluetouchpaper.org/2007/07/18/economics-of-tor-performance/.

[Mur11] Steven J. Murdoch. Comparison of tor datagram designs. Technical report, The Tor Project, November 2011.

[Pil13] Ed Pilkington. Declassified nsa files show agency spied on muhammad ali and mlk, September 2013. https://www.theguardian.com/world/2013/sep/26/nsa-surveillance-anti-vietnam-muhammad-ali-mlk.

[Pos80] J. Postel. User Datagram Protocol. RFC 768 (INTERNET STANDARD), August 1980.

[Pos81a] J. Postel. Internet Protocol. RFC 791 (INTERNET STANDARD), September 1981. Updated by RFCs 1349, 2474, 6864.

[Pos81b] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[RG09] Joel Reardon and Ian Goldberg. Improving Tor using a TCP-over-DTLS tunnel. In *Proceedings of 18th USENIX Security Symposium (USENIX Security 09)*. USENIX Association, August 2009.

[RM12] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012. Updated by RFCs 7507, 7905.

[Ros12] Jim Roskind. Quic: Design document and specification rationale. Technical report, Google, April 2012. https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit.

[Ste07] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), September 2007. Updated by RFCs 6096, 6335, 7053.

[TG10] Can Tang and Ian Goldberg. An improved algorithm for Tor circuit scheduling. In Angelos D. Keromytis and Vitaly Shmatikov, editors, *Proceedings of the 2010 ACM Conference on Computer and Communications Security (CCS 2010)*. ACM, October 2010.

[TS11] Florian Tschorsch and Björn Scheuermann. Tor is unfair - and what to do about it. In *LCN '11: 36th IEEE Conference on Local Computer Networks*. IEEE, October 2011.

[TS12] Florian Tschorsch and Björn Scheuermann. How (not) to build a transport layer for anonymity overlays. In *Proceedings of the ACM Sigmetrics/Performance Workshop on Privacy and Anonymity for the Digital Economy*, June 2012.

[TS13] M. Tuexen and R. Stewart. UDP Encapsulation of Stream Control Transmission Protocol (SCTP) Packets for End-Host to End-Host Communication. RFC 6951 (Proposed Standard), May 2013.

[Vie08] Camilo Viecco. Udp-or: A fair onion transport design. Technical report, July 2008. `https://www.petsymposium.org/2008/hotpets/udp-tor.pdf`.

[WT99] Alma Whitten and J. D. Tygar. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.

Appendix A

quux.h

Listing A.1: quux.h

```
#ifndef QUUX_API_H_
#define QUUX_API_H_

#ifdef __cplusplus
class quux_listener_s;
class quux_peer_s;
class quux_stream_s;
typedef class quux_listener_s* quux_listener;
typedef class quux_peer_s* quux_peer;
typedef class quux_stream_s* quux_stream;
extern "C" {

#else
typedef struct quux_listener_s* quux_listener;
typedef struct quux_peer_s* quux_peer;
typedef struct quux_stream_s* quux_stream;
#endif /* __cplusplus */

typedef void (*quux_connected)(quux_peer);
typedef void (*quux_cb)(quux_stream);

/*
 * Callbacks are triggered once when IO becomes actionable,
 * at which point no callback will be triggered until
 * the read/write operation on that stream has returned 0.
 *
 * A quux_stream begins in triggered state for both read/write,
 * so no further callbacks will be made.
 * However IO can still be attempted (albeit may return 0).
 */
```

```
struct event_base;

/**
 * Initialise the module, with your own libevent loop being used.
 *
 * EVLOOP_ONCE *must* be used in the event_base_loop call,
 * otherwise quux's internal time cache will start to go stale.
 */
void quux_event_base_loop_init(struct event_base*);

void quux_set_peer_context(quux_peer, void* ctx);
void* quux_get_peer_context(quux_peer);

void quux_set_stream_context(quux_stream, void* ctx);
void* quux_get_stream_context(quux_stream);

/**
 * Start a listener for new streams on IPv4 addr.
 *
 * quux_connected cb is called with the peer when a fresh client connects.
 *
 *
 * TODO: error if there is already a server listening on ip:port
 *
 * TODO: quux_set_connected_cb instead
 */
quux_listener quux_listen(const struct sockaddr* addr, quux_connected cb);

/**
 * Returns a handle representing an IPv4 connection to the peer.
 *
 * This will also start the crypto handshake in the background.
 *
 * WARNING: Incomplete:
 * TODO: FIXME: fail to connect if cert doesn't match hostname UTF8.
 */
quux_peer quux_open(const char* hostname, const struct sockaddr* addr);

/**
 * Create a new stream over the connection.
 */
quux_stream quux_connect(quux_peer peer);

/**
 * Nb. If the accept callback is not installed then incoming streams will be rejected.
 */
void quux_set_accept_cb(quux_peer, quux_cb quux_accept);

void quux_set_readable_cb(quux_stream, quux_cb quux_readable);
void quux_set_writeable_cb(quux_stream, quux_cb quux_writeable);

/**
 * When either side has decided to both stop reading and stop writing data,
 * this function will be called.
 *
 * The stream handle is still valid at this point.
 *
 * quux_free_stream should be called to free the memory.
 */
void quux_set_closed_cb(quux_stream, quux_cb quux_closed);

quux_peer quux_get_peer(quux_stream);

/**
 * 1 if the stream is fully closed, 0 otherwise.
 */
int quux_stream_status(quux_stream stream);

/**
 * Pass up to 'count' octets from 'buf' to the stream for send.
 *
 * Returned amount tells us how much data was transfered.
 * 0 indicates that no data could be written at this time, but the callback has been re-registered.
 * Call 'quux_write_stream_status' to find out if the stream is no longer writeable.
 *
 * The initial behaviour will be that once quux_read_close(); quux_write_close(); have been called,
 * it's at the discretion of the impl to wait as long
 * as necessary to receive acks for data before tearing down.
 *
 * At some point more functions could be added to query
 * the status of buffered data and force remove if needed.
 */
size_t quux_write(quux_stream stream, const uint8_t* buf, size_t count);

/**
 * Indicate that we don't want to write any additional data to the stream.
 */
void quux_write_close(quux_stream stream);

/**
 * 1 if the stream is fully closed, 0 otherwise.
 */
int quux_write_stream_status(quux_stream stream);

/**
 * Read up to 'count' octets from the stream into 'buf'
 *
 * Unlike quux_read, this will not consume the data from the stream,
 * so a subsequent peek or read will return the same data.
 *
 * Returned amount tells us how much data was transfered.
 * 0 indicates that no data could be read at this time, but the callback has been re-registered.
 * Call 'quux_read_stream_status' to find out if the stream is no longer readable.
 */
size_t quux_peek(quux_stream stream, uint8_t* buf, size_t count);

/**
 * Read up to 'count' octets from the stream into 'buf'
 *
 * Returned amount tells us how much data was transfered.
 * 0 indicates that no data could be read at this time, but the callback has been re-registered.
 * Call 'quux_read_stream_status' to find out if the stream is no longer readable.
 */
size_t quux_read(quux_stream stream, uint8_t* buf, size_t count);

/**
 * If 'count' octets are contiguously readable from the stream,
 * return a pointer to those octets.
 *
 * The pointer *must* be used before any further QUIC operations
 * and before the function returns, or it can become invalid.
 *
 * This function should only be used where the performance
 * overhead of memcpy might matter.
 *
 * NULL indicates that the requested amount is not available at the moment.
 * This function will not result in callbacks being reregistered in that case.
 *
 * quux_peek or quux_read should be used instead if NULL is returned,
 * since the data may be available, just not in a contiguous buffer.
 *
 * Call quux_read_consume afterwards to remove the data from input.
 */
uint8_t* quux_peek_reference(quux_stream stream, size_t count);

/**
 * Consume up to 'count' bytes from the stream input,
 * or the entirety if there was less than that amount available to read.
 */
void quux_read_consume(quux_stream stream, size_t count);

/**
 * Indicate that we don't want to read any additional data from the stream.
 */
void quux_read_close(quux_stream stream);

/**
 * 1 if the stream is fully closed, 0 otherwise.
 */
```

```
int quux_read_stream_status(quux_stream stream);

/**
 * Fully close the stream and free its memory.
 *
 * After this point, the handle will point to invalid memory and must not be used.
 */
void quux_free_stream(quux_stream stream);

/**
 * Close a connection ungracefully and free its memory.
 *
 * TODO: FIXME: Beware, this is currently incomplete and should not be used.
 */
void quux_close(quux_peer peer);

/**
 * Stop accepting connections
 *
 * TODO: FIXME: Beware, this is currently incomplete and should not be used.
 */
void quux_shutdown(quux_listener server);

/**
 * Run this just after libevent wait wakes up.
 *
 * It updates the approximate time internal to QUIC.
 */
void quux_event_base_loop_before(void);

/**
 * Run this just before going into libevent wait.
 *
 * It sends any packets that were generated in the previous event loop run.
 */
void quux_event_base_loop_after(void);

#ifdef __cplusplus
}
#endif

#endif /* QUUX_API_H_ */
```

Appendix B

Chutney experiment

Listing B.1: experiment.2.sh

```
#!/bin/bash

set -x
set -e

branch=$1
network=$2
loss=$3
extra=$4

NEWARK=45.79.174.25

echo experiment version: 3 branch: $branch network: $network loss: $loss extra: $extra

# clear out any cached state from previous runs
./configure-hosts.sh $network

cd $HOME/chutney

ssh root@$NEWARK "cd chutney; CHUTNEY_TOR=/root/$branch/src/or/tor ./chutney start networks/$network"

tc qdisc del dev eth0 root netem || true
CHUTNEY_TOR=/root/$branch/src/or/tor ./chutney start networks/$network

# minimum sleep before anything useful could happen
sleep 45

# just one run to set up a circuit
./chutney verify networks/$network
```

```
# packet loss
if [[ "$loss" && "$loss" != "0" && "$loss" != "0%" && "$loss" != "0.0%" && "$loss" != "0.000000%" ]];
then
    ssh root@$NEWARK "tc qdisc add dev eth0 root netem loss $loss"
fi

# clear network stats
nstat -r >/dev/null

# 32 MiB
CHUTNEY_DATA_BYTES=33554432 ./chutney verify networks/$network

# dump network stats difference
nstat

# get a rough feel for how cpu intensive it was
uptime

# TODO: put these in a trap exit
ssh root@$NEWARK "tc qdisc del dev eth0 root netem || true; \
        cd chutney; ./chutney stop networks/$network; killall -9 tor || true"
./chutney stop networks/$network
killall -9 tor || true

echo fin
```

Appendix C

Shadow experiment

Listing C.1: Dockerfile

```
FROM fedora:22

RUN dnf install -y git clang cmake make gcc-c++ xz igraph-devel glib2-devel llvm-devel file tar \
  automake golang zlib-devel

RUN useradd shadow
USER shadow

WORKDIR /home/shadow
RUN git clone https://github.com/aliclark/shadow.git

WORKDIR /home/shadow/shadow
RUN ./setup build
RUN ./setup install
ENV PATH $PATH:/home/shadow/.shadow/bin

WORKDIR /home/shadow
RUN git clone https://github.com/aliclark/shadow-plugin-tor.git

RUN mkdir /home/shadow/shadow-plugin-tor/build
WORKDIR /home/shadow/shadow-plugin-tor/build

RUN curl -O https://www.openssl.org/source/openssl-1.0.1h.tar.gz
RUN echo 9d1c8a9836aa63e2c6adb684186cbd4371c9e9dcc01d6e3bb447abf2d4d3d093 openssl-1.0.1h.tar.gz \
        | sha256sum -c --strict
RUN tar xaf openssl-1.0.1h.tar.gz
WORKDIR /home/shadow/shadow-plugin-tor/build/openssl-1.0.1h
RUN ./config --prefix=/home/shadow/.shadow shared threads enable-ec_nistp_64_gcc_128 -fPIC
RUN make depend
RUN make -j $(nproc)
RUN make install_sw

WORKDIR /home/shadow/shadow-plugin-tor/build
RUN curl -O https://cloud.github.com/downloads/libevent/libevent/libevent-2.0.21-stable.tar.gz
RUN echo 22a530a8a5ba1cb9c080cba033206b17dacd21437762155c6d30ee6469f574f5 \
        libevent-2.0.21-stable.tar.gz | sha256sum -c --strict
RUN tar xaf libevent-2.0.21-stable.tar.gz
WORKDIR /home/shadow/shadow-plugin-tor/build/libevent-2.0.21-stable
RUN ./configure --prefix=/home/shadow/.shadow
RUN make -j $(nproc) install

WORKDIR /home/shadow/shadow-plugin-tor/build
RUN git clone https://github.com/aliclark/libquic
RUN mkdir libquic/build
WORKDIR /home/shadow/shadow-plugin-tor/build/libquic/build
RUN cmake .. -DCMAKE_BUILD_TYPE=Release
RUN make -j $(nproc)

WORKDIR /home/shadow/shadow-plugin-tor/build
RUN git clone https://github.com/aliclark/tor
WORKDIR /home/shadow/shadow-plugin-tor/build/tor
RUN ./autogen.sh
RUN ./configure --disable-transparent --disable-asciidoc \
        CFLAGS="-fPIC -fno-inline -I../libquic/src/quux" \
        --with-libevent-dir=`readlink -f ~`/.shadow --with-openssl-dir=`readlink -f ~`/.shadow
RUN make -j $(nproc) || echo \
        "will fail to link quux due to hard-coded static lib path, but doesn't matter here"

RUN mkdir /home/shadow/shadow-plugin-tor/build/main
WORKDIR /home/shadow/shadow-plugin-tor/build/main
RUN CC=/usr/bin/clang CXX=/usr/bin/clang++ cmake ../.. \
        -DTOR_VERSION_A=0 -DTOR_VERSION_B=2 -DTOR_VERSION_C=7 -DTOR_VERSION_D=6
RUN make -j $(nproc) install

WORKDIR /home/shadow/shadow-plugin-tor/resource

# Uses over 32G memory (about 35) for 60m simulation
RUN tar -xf shadowtor-toy-config.tar.xz
#
# RUN tar -xf shadowtor-minimal-config.tar.xz

WORKDIR /home/shadow/shadow-plugin-tor/resource/shadowtor-toy-config
# WORKDIR /home/shadow/shadow-plugin-tor/resource/shadowtor-minimal-config

# TODO: Disable logging in the torrc files first?

# TODO: pass out shadow.data and shadow.log (as a tar.gz on stdout?)
# Or copy onto a volume?
#
# ENTRYPOINT shadow-tor -y -w $(expr $(nproc) - 1)
```