MASTER'S THESIS

Analysis of UDP-based Reliable Transport using Network Emulation

Andreas Vernersson
2015

Master of Science in Engineering Technology
Computer Science and Engineering

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering

Abstract

The TCP protocol is the foundation of the Internet of yesterday and today. In most cases it simply works and is both robust and versatile. However, in recent years there has been a renewed interest in building new reliable transport protocols based on UDP to handle certain problems and situations better, such as head-of-line blocking and IP address changes.

The first part of the thesis starts with a study of a few existing reliable UDP-based transport protocols, SCTP which can also be used natively on IP, QUIC and uTP, to see what they can offer and how they work, in terms of features and underlying mechanisms.

The second part consists of performance and congestion tests of QUIC and uTP implementations. The emulation framework Mininet was used to perform these tests using controllable network properties. While easy to get started with, a number of issues were found in Mininet that had to be resolved to improve the accuracy of emulation.

The tests of QUIC have shown performance improvements since a similar test in 2013 by Connectify, while new tests have identified specific areas that might require further analysis such as QUIC's fairness to TCP and performance impact of delay jitter.

The tests of two different uTP implementations have shown that they are very similar, but also a few differences such as slow-start growth and back-off handling. A previously identified problem with increasing queuing-delay could be replicated with both uTP implementations but new experiments show that a small amount of packet loss seems to prevent this problem.

This work has shown how Mininet can be used to build test systems to analyze UDP-based transport protocol implementations.

Contents

Chapter 1

Introduction

The Transmission Control Protocol (TCP) is the foundation of the Internet of yesterday and today. In most cases it simply just works and is both robust and versatile. However, in recent years there has been a renewed interest in building new reliable transport protocols based on the unreliable User Datagram Protocol (UDP).

In this thesis, we will first study three reliable UDP-based protocols to learn what problems and situations they are trying to handle better and how they are different from TCP.

Stream Control Transport Protocol (SCTP) is a generic transport protocol that designed to run directly over IP[1], but can also be used over UDP when the operating systems that lacks a kernel implementation of SCTP[2]. Thereby, a user space implementation of SCTP can be used.

Google's Quick UDP Internet Connections (QUIC) is used for transporting web requests and responses[3].

uTorrent Transport Protocol (uTP) which is used by BitTorrent for peer to peer file sharing[4].

Looking at SCTP, QUIC and uTP, several features that are not widely possible with TCP emerge. For example QUIC connections can handle IP address changes[3], which are not uncommon when smart phones switch between Wi-Fi and mobile networks, and uTP that can transfer files using low bandwidth priority[4].

The second part of this thesis examines QUIC and uTP in more detail to test actual protocol implementations using network emulation. These tests are performed to evaluate the implementations' performance characteristics in different network situations, and their congestion handling to see how they react to concurrent TCP connections.

The tests were performed using Mininet[5], a software based network emulator, which makes it possible to test implementations using different network properties in a controlled environment.

1.1 Background

The background of this thesis was to learn more about new UDP-based reliable transport protocols and the limitations of TCP. Not for implementing or improving a specific application, but to be able to consider alternative protocols when developing new applications in the future. Depending on the situation and the set of requirements, there might be a more suitable protocol than TCP.

The tests are performed to evaluate the implementations, both to gain better experience and understanding of them but also to create a test system that can be useful to test other implementations in the future. The performance characteristics in different network situations and comparison with TCP are important decision factors when selecting an implementation. To check their congestion handling is important to avoid causing unintended effects for other users or the networks.

QUIC was selected because it's a new and relatively complex protocol that has already been deployed by Google servers and in Chrome browser[3][6], and because it has not seen much public analysis yet. uTP was selected because of its unique low priority congestion control and it's wide usage[7] and that multiple implementations exists and can be compared[8][9].

1.2 Problem description and goals

The goal of this thesis is to examine the design of the protocols SCTP over UDP, QUIC and uTP and to test the implementations of QUIC and uTP to evaluate their network performance and congestion handling.

In particular, the thesis tries to answer the following questions:

- What limitations of TCP can reliable UDP-based protocols handle better?
- Which features do reliable UDP protocols offer compared to TCP and how do they work?
- How can Mininet be used to test reliable UDP implementations?
- How does the QUIC implementation perform compared to TCP?
- What differences are there between the popular uTP implementations and how do they perform compared to TCP?

1.3 Demarcations

To limit the scope of the thesis and keep time constraints, the following was not included.

- Detailed review of the protocol design or implementations.
- Complete evaluations of the implementations, for example their compatibility, scalability or security.
- The effects when emulating corruption and reordering to network packets.

1.4 Thesis outline

Chapter 2 begins with a short refresher on reliable transport protocols, a few relevant TCP mechanisms, why UDP is necessary and a few words about delays and fairness. Thereafter follows the study of three existing reliable UDP transport protocols to see what problems they are trying to solve and how they work, including related work. The chapter ends with an introduction to Mininet, the network emulator used to test and measure different aspects of the implementations.

Chapter 3 describes the methodology of the work.

Chapter 4 details how Mininet was used, including verification that the network emulation works as expected. Problems and solutions that were encountered during the process are also described.

Chapter 5 and 6 contains the actual tests of two reliable UDP protocols, QUIC and uTP. First describing how the tests were performed, followed by results and analysis.

Chapter 7 contains the concluding words of this thesis; about the protocols, implementations and Mininet.

Appendix A lists details about the test system, hardware, software and tested versions. Appendix B describes the abbreviations used.

Chapter 2

Theory

The Internet protocol suite is typically described using five layers, which include a *transport protocol* layer. The purpose of the transport layer is to transport application-layer messages between endpoints. This will provide a logical communication channel between applications running on different endpoints. The service level that this channel provides is widely different for the two main transport protocols UDP and TCP[10]. A bare minimum is the separation of messages from different applications through the use of abstract ”ports”, but may also include error control and flow control.[11]

The service level offered by UDP is essentially the minimum level, which is also known as a *connectionless, unreliable transport protocol* since UDP makes little to improve the shortcomings of the underlying network layer. UDP does not try to detect or retransmit lost packets, check that packets arrive in the original order or to control the packet flow[12].

TCP on the other hand is the transport protocol which has essentially defined the meaning of a *connection-oriented, reliable transport protocol.* Such a reliable protocol consists of multiple services, summarized in table 2.1. Essentially, TCP detects lost packets and automatically resends them, divides the application's data into multiple packets and combines them in the correct order. It should also match the sending rate with the receiver's capacity to process data. It needs to detect when the network experiences congestion and make sure multiple users will get a fair bandwidth share[11].

Service	**Description**
Separation	Data is separated through usage of ”ports”.
Connection-oriented	A virtual connection is established between two hosts.
Corruption	Verification that data is unmodified.
Resending	Lost or corrupted packets are retransmitted.
Duplication	Duplicated packets are discarded.
Framing	Large data is divided into multiple packets and later reassembled.
Ordering	Data is delivered in the correct order.
Flow control	Adapts sending rate to receiver’s capacity.
Congestion control	Network congestion is detected and avoided.
Fairness	Multiple concurrent connections will get a fair share of bandwidth.

Table 2.1: Requirements of a reliable transport protocol

In practice, since TCP is so dominating, new reliable transport protocols are expected to behave similarly to TCP and not claim a larger bandwidth share, largely to avoid creating new congestion problems or downgrading the performance of TCP connections.

The service level for different transport protocols is displayed in table 2.2. Note that even unreliable protocols such as UDP include some but far from all of the requirements.

The UDP-based protocols can utilize the UDP services, separation and corruption, but this is not always done. uTP depends on UDP’s checksum for corruption and partially on UDP’s ports for separation. QUIC and SCTP over UDP make minimal use of the UDP services, using their own mechanisms instead. This will be explained in their respective sections.

---

[1] Typical underlying ”link layer”

Service	**UDP**[12]	**TCP**[13]	**SCTP**[1]	**QUIC**[3]	**uTP**[4]
Underlying layer[1]	IP	IP	IP or UDP[2]	UDP	UDP
Separation	Y	Y	Y	Y	Y
Connection-oriented	N	Y	Y	Y	Y
Corruption	Y	Y	Y	Y	Y
Resending	N	Y	Y	Y	Y
Duplication	N	Y	Y	Y	Y
Framing	N	Y	Y	Y	Y
Ordering	N	Y	Y	Y	Y
Flow control	N	Y	Y	Y	Y
Congestion control	N	Y	Y	Y	Y
Fairness	N	Y	Y	Y	Y
”Reliable transport”	N	Y	Y	Y	Y

Table 2.2: Service level for different transport protocols

2.1 A short history of TCP

The historical context is worth considering since the introduction of TCP in the 1970s. The networks of that time often had bandwidths that were measured in a Kbit/s and packet delays were normally in the seconds. The networks were not for public access and had relatively few numbers of users. The applications that used TCP were mainly text-oriented, such as remote computer login, file transfer and electronic mail[14].

TCP has been able to improve and stay dominant through the years. The original TCP specification did not include congestion control[15] and the early Internet experienced multiple ”congestion collapses” before the development of congestion avoidance and control[16] resulted in an updated TCP specification in 1998[13]. The congestion control has also been improved, many variants have been researched and host systems now use several different algorithms such as ”New Reno”, ”Cubic” and ”Compound TCP’.

Today’s bandwidths are magnitudes higher and latencies much lower but still limited by distance and the speed of light. The extension mechanism of TCP options has enabled gradual improvement while maintaining compatibility. Specifically ”window scaling” that increases the maximum window size from previously only 64 KB to 1 GB has been essential to reach higher speeds over long distances and ”timestamps” for better round-trip time estimation[17]. Another important TCP option is the ”Selective Acknowledgments” (SACK) to improve the effectiveness of handling packet loss[18].

TCP is designed to transport a single byte stream not preserving the individual message sizes. Today's application protocols are less based on text that can be parsed without preserved message lengths and more on binary messages that do require preservation of message-boundaries, which require applications to include message lengths. This data overhead might not be large, but since TCP doesn't understand these boundaries it can lead to unfavorable "packetization" or buffering. This problem is partly solved by Nagle's algorithm that tries to avoid unnecessary division of data in small packets.

Several modern applications utilize a single connection for doing multiple tasks at once, such as Hypertext Transfer Protocol version 2 (HTTP/2) that can request and download multiple files simultaneously[19]. Multiplexing is used for example to reduce redundancy of using multiple connections that use the same HTTP headers or encryption handshake. Such multiplexing of streams in TCP leads to a problem called head-of-line blocking. When a single packet is lost other streams not related to the loss are blocked until the lost packet is retransmitted. This introduces unnecessary latency in the other streams. Since TCP uses a single byte-oriented stream it can't deliver data out of order to the application layer.

Other modern requirements such as increased level of data protection and privacy have mostly been solved on top of TCP using Transport Layer Security (TLS)[20] or in the application protocol, such as SSH[21]. One exception is the QUIC protocol, which has integrated encryption and will be described later in this chapter.

The limitations of TCP yet to be solved are probably not critical enough to require large and possibly non-backward compatible changes. Since TCP is both governed by IETF standards and implemented at the center of operating systems, any changes to the protocol would take a very long time to gain widespread support.

2.2 TCP mechanisms

Many of TCP's mechanisms and algorithms that have evolved over the years are also used in other reliable transport protocol and implementations. This is described in much more detail in for example[11], but here are brief descriptions of the most relevant parts for this thesis.

To establish a new connection, TCP typically uses a 3-way handshake using different flags in the TCP header. The client sends a packet with the SYN-flag, which the server acknowledges by a packet with SYN- and ACK-flags which is finally acknowledged by the client with a packet with the ACK-flag. Other flags are used to signal the end of a connection.

To support separation and allow multiple connections and applications, TCP uses the 4-tuple (IP source address, IP destination address, TCP source port, TCP destination port) as a unique identifier for each connection on a host. The IP addresses are already present in the IP header and the port numbers are included in the TCP header. If either host changes its IP address, there is no possibility to tell the other of the new address to continue established connections. A single TCP connection cannot utilize multiple interfaces with different IP addresses to improve performance or availability in case of failure.[13]

Accidental corruption of packets is detected by including a checksum in each packet of the TCP header and data, which the receiver will verify. Corrupted packets are discarded and the sender has to resend the data later.

TCP uses sequence numbers to keep track of the transmitted data. These sequence numbers refer to byte offsets, starting at a randomized value. Sequence numbers allow the receiver to reassemble packets in the correct order, discard duplicate packets and send acknowledgment numbers with information about the data received. The receiver will only tell the sender the highest sequence number of contiguously received data. The sender has to keep transmitted data in a ”sliding window” until their acknowledgement has been received. To retransmit lost packets, the sender must first realize which packets are missing, either indirectly by receiving multiple ACKs for the same sequence number, duplicate ACKs, or after waiting enough time for the acknowledgment (timeout).

When the TCP option ”Selective ACK” is used, the receiver can also specify multiple ranges of data that has been received instead of just the first contiguous range. Surprisingly SACKs are only advisory so a sender can’t remove data from the send-window, because a receiver is allowed to change its mind and later ”un-acknowledge” the data. Due to size limitations of TCP options and the SACK format requiring 8 bytes per range, only 3 or 4 different ranges can normally be specified.

A window-based rate control of sending speed allows a window-size amount of data to be transmitted before having to wait a round-trip time (RTT) to receive an update on the window-size. This effectively limits sending rate R from window-size W packets, packet size S bits and RTT seconds to $R = W * S/RTT$

TCP uses window-based flow control to match the sending rate with the receiver’s capacity. The receiver uses the ”window-size” header field to indicate the number of bytes available in its receive buffer. The sender cannot transmit more data than the latest advertised window-size.

TCP’s congestion control also uses a window-based rate control, where the congestion window-size, CWND, is the number of outstanding bytes that can be transmitted before acknowledgement. Variants of TCP's congestion control calculate CWND in different ways but two common phases are *slow-start* and *congestion-avoidance*. In *slow-start* CWND starts very small but while receiving ACKs it can grow exponentially, thus quickly increasing send rate. Slow-start is kept until CWND reaches a certain level, slow-start threshold, and then changes to *congestion avoidance*. Here CWND grows only linearly while receiving ACKs, but is reduced exponentially when packet loss or congestion is detected. This is called Additive Increase Multiplicative Decrease (AIMD) and is central for TCP's congestion control.

To combine both flow control and congestion control, the sending rate is determined by the minimum of advertised window-size and congestion window-size.

2.3 Why UDP

When building a new reliable transport protocol, the logical choice would be to use IP as the underlying network layer. This however doesn't work in deployed networks and systems for two main reasons.

Firstly, endpoints in both residential and mobile networks are now almost always using NAT with IPv4. In the future this will be even more common with the limitations of IPv4 space. This means that endpoints can only use transport protocols supported by the NAT-device if they want to talk bidirectionally with an ”outside” peer. The only viable transport protocols that work through NAT, firewall and other ”middle boxes” are TCP and UDP[3].

Secondly, for applications to send own IP-packets they would either need to use a non-standard operating system feature or implement the transport protocol as a kernel-mode driver, similar to TCP or UDP. Both of these options require special privileges which can be problematic. For mobile applications neither option is widely possible.

One downside of UDP is the requirement of a UDP header: source port, destination port, length and checksum. This information can be redundant for other transport protocols since identifying connections can be done by other means than the typical 4-tuple. Length is already conveyed by the IP-layer and the checksum can only detect accidental packet modifications.

Another downside with UDP when used through NAT is the short timeout of UDP bindings compared to TCP. The recommended minimum timeout value is 120 seconds for UDP[22] compared to 2 hours and 4 minutes for TCP[23].

The upside is that applications can deploy and use their own protocols and implementations, independently of operating system limitations or Internet standard processes. They also have a chance of working across NAT-boundaries and other existing infrastructure[2][3].

2.4 Overview of UDP-based protocols

The first sections offer an introduction to the requirements a reliable UDP-based protocol has to implement. The overview of TCP shows some of its limitations that UDP-based protocols can solve.

Keeping the basic reliability requirements, a UDP-based protocol is relatively free to innovate and implement new features. A summary of such features present in the studied protocols are shown in table 2.3 and will be explained in more detail in the following sections.

Feature	**TCP**[13]	**SCTP**[1]	**QUIC**[3]	**uTP**[4]
Unordered delivery	N	Selectable	N	N
Partial reliable delivery	N	Optional[24]	N	N
Message boundary preservation	N	Y	N	N
Low Priority Congestion	Optional[25]	Optional[25]	N	Y
Selective ACKs	Optional[18]	Y	Y	Y
Large window-size	Optional[17]	Y	Y	Y
Forward error correction	N	N	Y	N
Multiple streams	N	Y	Y	N
Change interface or IP address	N	Y	Y	N
Multiple interfaces or IP addresses	N[2]	Y	N	N
Chunked packet format[3]	N	Y	Y	N
Data encryption	N	N	Y	N
SYN-flood protection	Optional[27]	Y	Y	N
Malicious ACK protection	N	N	Y	N
Socket API	Y	Y[4]	N	N

Table 2.3: Features in the studied reliable transport protocols

---

[2] Multipath TCP is a new experimental standard[26].

2.5 Delays, BDP and fairness

Kurose and Ross[10] describe the delays on intermediate network nodes as the sum of four components:

$$nodal\_delay = processing + queuing + transmission + propagation \tag{2.1}$$

- Processing is the time it takes for handling a packet, before entering the send queue.
- Queuing is the time the packet waits in a queue until transmission starts. Depends on the current utilization of the link and indirectly on bandwidth.
- Transmission is the time it takes to send a packet to the ”wire”. Depends on packet size and bandwidth.
- Propagation is the time it takes for a packet to reach the receiving node. Depends on propagation speed and length of the ”wire”.

For a network where a packet passes through multiple nodes, the total delay or ”actual latency” can be described as the sum of all intermediate nodal delays. When using emulated networks the parameter called ”delay” is used to simulate the propagation delay, the minimum delay all packets are subject to.

One simple but important relationship is the calculation of transmission time:

$$transmission\_time = packet\_bits/bandwidth \tag{2.2}$$

This relationship is also used in the Bandwidth Delay Product, BDP, which given bandwidth and time will tell how many bits can be transmitted. BDP can also calculate the required window size or queue size to avoid waiting for acknowledgement packets, using round-trip-time (RTT) as delay:

$$BDP = bandwidth * delay \tag{2.3}$$

---

[3] Whether each packet can contain multiple data chunks

[4] If SCTP is implemented in the operating system

Regarding fairness, that multiple simultaneous protocols get a fair share of bandwidth. For N flows through a bottleneck link with rate R, each should use an average rate of R/N. There are many ways of testing this property. One is to visualize the individual bandwidths over time.

Another is the Jain's Fairness Index to quantify fairness into an intuitive number, where 1.0 means that all of the flows get exactly equal share and 0.5 means that only 50% of the flows get an equal share. It can be calculated for N flows where flow i uses x_i bandwidth using equation 2.4.[28]

$$F = \frac{(\sum_1^N x_i)^2}{N * \sum_1^N x_i^2} \tag{2.4}$$

Individual tests of bandwidth share can be significantly different, so in practice many runs must be made and statistically analyzed to get a useful result.

Other methods of testing fairness include ”TCP Friendly Rate Control” equation[29] or ”relative fairness” to compare transfer rate for different loss probabilities[11][p.768].

2.6 SCTP

The Stream Control Transport Protocol (SCTP) was developed to transport telephone signaling over IP, but is a generic reliable transport protocols with some unique features. The original standard from 2000[30] has been updated in 2007[1] and multiple extensions exist. SCTP can also be used over UDP using a simple encapsulation format [2]. The following brief description is these specifications.

A SCTP connection is established through a four-way handshake, using the sequence of INIT, INIT-ACK, COOKIE-ECHO and COOKIE-ACK. This is designed to avoid SYN-flooding attacks with the wrong source-IP addresses, similar to a previous cookie mechanism[31]. Essentially the server can avoid allocating resources when receiving INIT. Instead the server sends INIT-ACK containing a specially constructed cookie value that the client has to include in COOKIE-ECHO. The server can verify that the cookie is correct and only after knowing the client's source IP to be true, allocate connection resources and finish the handshake with COOKIE-ACK.

SCTP is one of few transport protocols where a single connection can utilize multiple interfaces and IP-addresses to improve fault tolerance (reachability when a link fails). This is called multi-homing, and each host must advertise their IP-addresses where they can be reached. A host will continuously probe the other host's addresses to determine current reachability and automatically select a primary path. In case of SCTP with NAT, the hosts can be unaware of their external IP-addresses which require the Dynamic Address Reconfiguration extension[32] where the source addresses of incoming packet is used instead.

To support multi-homing, connections are not primarily identified by the 4-tuple as TCP, but by 32-bit verification tags established during the handshake. Each direction of the connection uses a unique verification tag and all SCTP packets must contain the correct one.

SCTP supports multiple streams in each connection, called multi-streaming. Each stream transports a separate sequence of application messages, where individual lengths and inter-message ordering are preserved. Each stream is independent of the others so that packet loss in one stream does not affect others, as messages are delivered separately to the application to avoid head-of-line blocking. Application messages can also select unordered delivery, which are delivered as soon as they arrive to the application layer.

By default, chunks from different streams can be sent together in a packet, called chunk bundling, either by allowing a certain delay for filling packets or without introducing extra delays. An application can choose to avoid chunk bundling, but in case of congestion the implementation is still allowed to bundle.

Similar to TCP and UDP, SCTP also uses 16-bit source and destination port numbers to implement separation, and a 32-bit checksum to detect network corruption. SCTP over UDP still includes SCTP own ports and checksum in addition to the UDP ports and checksum when used over UDP. This allows multiple SCTP applications to use the same SCTP port by using different UDP ports.

Because the SCTP packet format is based on chunks with type, length and data fields, along with guidelines on how to introduce new chunk types, SCTP can be extended while maintaining backward compatibility.

An example of such extension that introduces new chunk types is the Partial Reliability Extension[24] which enables the sender to control how many (if any) retransmissions it will try before telling the client to ignore lost messages and continue with new messages. This feature is wanted for services that need a reliable control channel along with data that quickly becomes obsolete, such as real-time audio and video.

2.7 QUIC

Quick UDP Internet Connections (QUIC) is a new protocol designed by Google that first was made public in 2012. It is still under development and experimentation, with an evolving design document[3], cryptography document[33] and packet specification[34] available online. QUIC is already supported by the Google's Chrome browser and services such as Google Search and Youtube[3].

QUIC aims to replace both TCP and TLS to transport HTTP/2, the next generation HTTP protocol. The goals of QUIC are many but essentially Google wants a protocol that can be deployed on today's Internet that reduces latency and also solves problems with multiple streams over a single TCP connections. Other goals include improved latency and efficiency for mobile platforms, privacy comparable to TLS and mitigations against denial of service attacks.[3, GOALS]

Lower latency is wanted to reduce the effects of round-trip time (RTT) when establishing new connections. By integrating TCP and TLS in a single protocol QUIC can avoid two sequential handshakes. QUIC can more importantly completely avoid round-trips, called 0-RTT connection latency. This can be done when a client already knows the server's public key so that it can send the "SYN" and proposed and encrypted session key in the first "Hello" packet. If the public key is still valid, the server can decrypt and establish this new connection directly or require an additional round trip to mitigate against source IP spoofing. The 0-RTT mechanism builds on previous ideas/works, including "TCP Fast open"[35] and "TLS Snap Start"[36]. See figure 2.7.1 for a brief connection establishment comparison between TCP/TLS and QUIC.

QUIC solves head-of-line blocking by supporting multiple streams that have individual flow control and are designed not to introduce unwanted dependencies. Individual QUIC streams can for example be decrypted independently. Multiplexing streams in TCP also leads to a bandwidth disadvantage compared to parallel connections, partly because a lost packet reduces all streams bandwidth and partly because multiple connection can increase the total bandwidth faster during slow-start. To compensate for this, QUIC's streams also have individual congestion control.

QUIC supports different congestion control algorithms, which are negotiated upon connection establishment. The default algorithm is based on TCP-Cubic and other variants based on are also going to be available.[34].

Other notable features in QUIC:

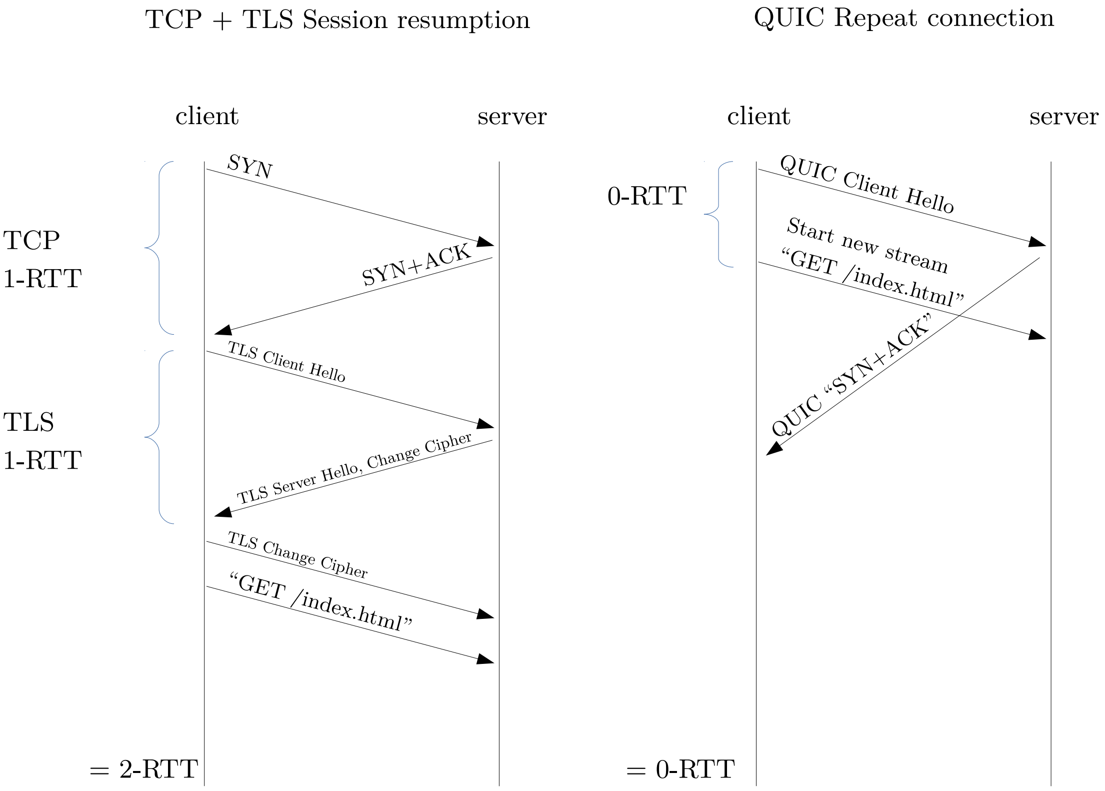


Figure 2.7.1: Best case TCP/TLS and QUIC connection establishment

- Connections can survive a change of source IP and port by including a Connection ID in the packet header.

- Uses forward error correction (FEC) to reduce the risk of retransmission latency when packet loss occurs by sending parity packets every N packets. If single packet in a ”packet train” is lost, the data can be recovered without any retransmission.

- Protection against malicious ACKs to make the sender send faster by requiring ACKs to include a proof that the packets were actually received.

Since QUIC supports that IP address changes, great care have been taken not to become vulnerable to sending too much data to the wrong recipient when forged source addresses are used by an attacker. This is done using protocol features where the server can require the client to prove it both can send and receive packets on the new IP. Such proofs are time limited and for repeat connections they could have expired.

2.7.1 QUIC Implementation

QUIC has been implemented as part of the Chromium project and is available as open source[5]. Unfortunately, no publicly available web server currently supports QUIC.

The Chromium implementation is written in C++ as in a extensive collection of classes and unit tests. Also available are command-line test tools that can be used to do simple HTTP-like requests and responses with QUIC, called test-server and test-client. Table 2.4 shows brief source code metrics that were captured using the ”Count Lines of Code” (CLOC) utility[6].

Component	Files	Classes	Lines of code	Lines of comment
Protocol	227	430	30042	6835
Protocol unit tests	129	219	27405	3679
Test tools	34	83	3367	659
Test tools unit tests	36	65	4627	687

Table 2.4: CLOC statistics for the QUIC implementation

2.7.2 Related work on QUIC

QUIC is such a new protocol that the only found study of QUIC is a blog post by the company Connectify in late 2013[37].

This test compares QUIC to HTTP in three different cases. All cases measure the time it takes to download a single 10 MB file with random content. This time measurement includes connection establishment, file request, file download and connection close. The elapsed time is converted to Mbit/s and is called ”goodput” to make comparisons with the bandwidth limit more intuitive.

$$goodput = file\_size\_bits/download\_time \tag{2.5}$$

The tests were made by using two computers connected via a hardware router running Linux/OpenWRT using Netem to simulate different network properties. Three different

---

[5] http://src.chromium.org/viewvc/chrome/trunk/src/net/quic/

[6] http://cloc.sourceforge.net

cases were tested and the results are shown in figure 2.7.2.

- Varying bandwidth, 2-70 Mbit/s symmetric bandwidths with 20 ms RTT and 0% loss.
- Varying network delay, 80-800 ms RTT with 10 Mbit/s symmetric bandwidth and 0% loss.
- Varying packet loss probability, 0-5% loss with 5 Mbit/s symmetric bandwidth and 160 ms RTT.

The results showed that QUIC were generally slower than HTTP in all cases except when there is more 1% packet loss. The logical explanation why it was faster in the loss case is QUIC's forward error correction, except for it was disabled in QUIC's code at the time. The discussion why it could be faster anyway was not presented.

In all cases QUIC would stay below 9 Mbit/s which was explained by QUIC's use of packet pacing which would only send one packet every 1 ms, which means 1000 packets/s. This limit transfer rate in the base case of timer resolution to:

$$1000 packet/s * 1500 bytes/packet * 8 bits/byte \cong 12 Mbit/s \tag{2.6}$$

The Connectify test received some feedback from the QUIC developers on QUIC's mailing list[7], which noted that FEC was disabled at the time and also that the congestion window size was temporary limited for avoiding or finding a bug.

---

[7] https://groups.google.com/a/chromium.org/forum/#!topic/proto-quic/xoooxrixCis

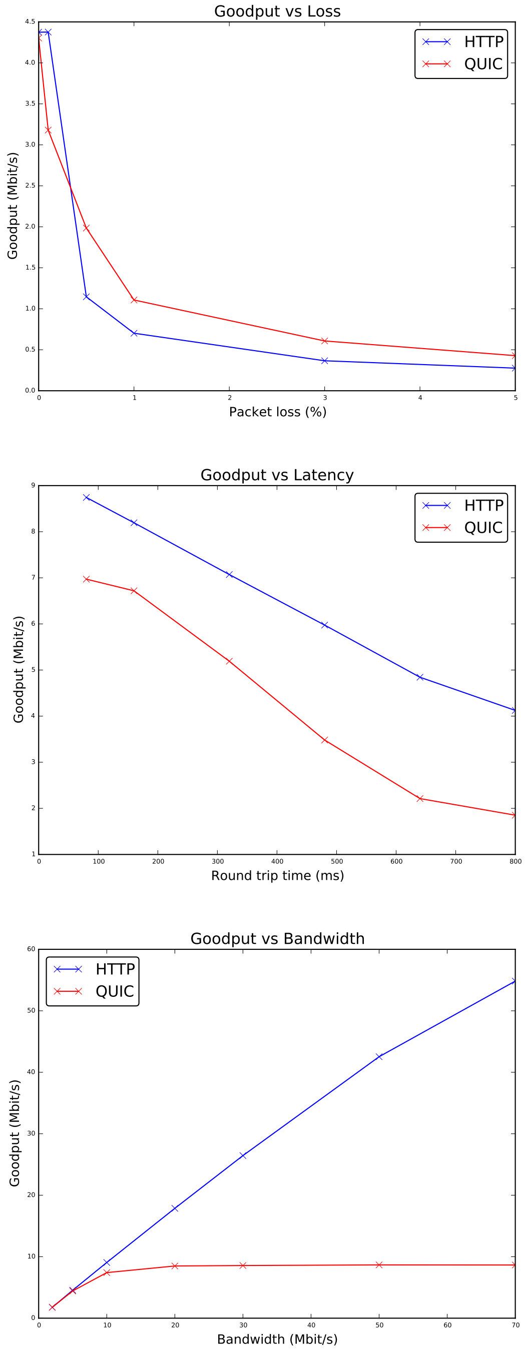


Figure 2.7.2: Connectify's QUIC/HTTP results with goodput (Y) compared to the varying parameter (X).

2.8 uTP

The uTorrent Transport Protocol (uTP) was released in 2009 to be used for the BitTorrent file sharing application protocol[4]. The congestion control algorithm in uTP was later standardized as Low-Extra-Delay BAckground Transport (LEDBAT)[38], and can be applied to other protocols such as TCP and SCTP. By some accounts uTP is used for between 13 and 20% of the traffic on the Internet[7].

The goal was to fully utilize bandwidth only when there is no other traffic usage. This was to minimize delays or freeze in interactive usage such as web browsing, real-time audio/video or gaming while running BitTorrent. These delays was caused by TCP filling the uplink's send-buffer, which in DSL and Cable modems often have seconds worth of buffer space. These seconds of delays turns out much worse because TCP slow-start often requires several round-trips to transfer the wanted data. This overly usage of send-buffer causing delays is called buffer-bloat.

uTP introduces both a new UDP-based reliable transport protocol and to avoid buffer-bloat a novel delay based congestion control algorithm. The algorithm measures one-way queuing-delay and adjusts the sending-rate proportionally to a target delay, the queuing delay that is ”allowed” to be introduced by uTP's own traffic.

The protocol uses regular mechanisms such as 3-way handshake to initiate connections, sequence-numbers, acknowledgements, receive window, congestion windows and round-trip-time estimation from ACKs and timeouts for packet resending.

Compared to TCP, large receive windows are supported without requiring a window-scale option[8] and SACKs are always supported. Sequence-numbers and ACKs refer to packet number, not byte number. This allows SACKs to use bit fields to mark which individual packets have been received, to effectively notify multiple missing packets.

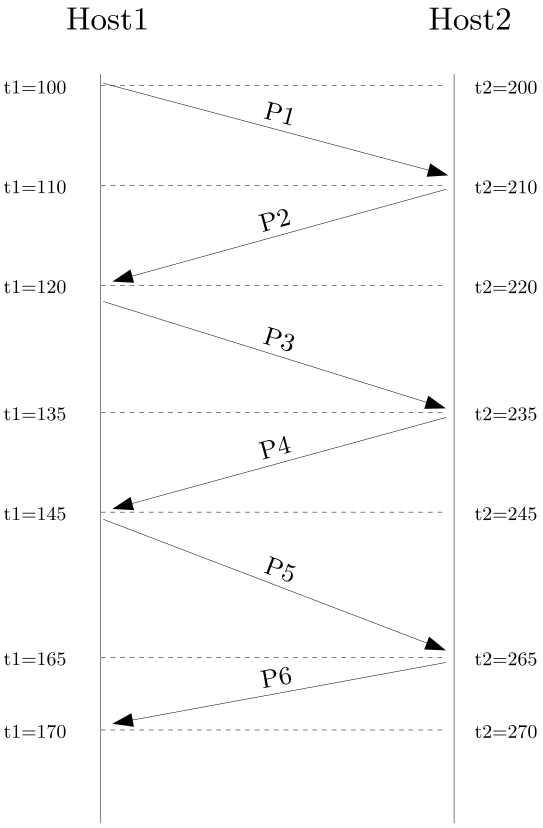
To enable the congestion algorithm, the protocol headers include a high resolution time-stamp when the packet is sent and a time-stamp difference. The time-stamp difference is calculated when receiving a packet between the current local time and the packet's time-stamp. This difference is then used in the next outgoing packet. The received difference values are stored in a sliding history list. The specification says that this history list should keep track of the minimum value, called *base_delay* for the last 2 minutes. The difference values individually are meaningless since the two hosts have different time references but by subtracting *base_delay*, actual one-way queuing delay samples can be extracted, called *our_delay*. This method is used to remove non-queuing delays caused

---

[8] Max window size for uTP is 4 GB compared to TCP without/with WSOPT 64 KB/1 GB

by processing, transmission and propagation which should stay roughly constant while the route stays the same.

Figure 2.8.1 shows how these time-stamps are calculated and propagated back. Host1 will learn from t_diff history that its uplink queuing delay is increasing. Host2 will see that its uplink queuing delay is constant. Host1 can also store sent differences and also understand that Host2's uplink looks good. The situation is simplified so receiving time and the next packet is sent on the same time value. The two hosts' clocks are also not drifting or wrapping around their max value, both of which requires special handling.



P	t	t_diff	recv_t	recv_t - t
P1	100	n/a	210	110
P2	210	110	120	-90
P3	120	-90	235	115
P4	235	115	145	-90
P5	145	-90	265	120
P6	265	120	170	-95

Host1 received t_diff history

	t_diff	our_delay
P2	110	110-110 = 0
P4	115	115-110 = 5
P6	120	120-110 = 10

base_delay = 110

Host2	t_diff	our_delay
P1	n/a	n/a
P3	-90	-90-(-90) = 0
P5	-90	-90-(-90) = 0

base_delay = -90

Figure 2.8.1: Example of measuring one-way queuing delays using uTP transmission time-stamps (t) and time-stamp differences (t_diff)

The congestion control changes the sending rate by updating the allowed window size (max_window) according the equations in figure 2.8.2. [4]

$$delay\_factor = (\mathbf{target\_delay} - cur\_delay)/\mathbf{target\_delay} \tag{2.7}$$

$$window\_factor = outstanding\_packet/max\_window \tag{2.8}$$

$$max\_window = max\_window + \mathbf{gain} * delay\_factor * window\_factor \tag{2.9}$$

Figure 2.8.2: uTP congestion control calculation

So depending if current delay is below or above **target_delay**, the window size is either increased or decreased. The size of the change depends both on how far off delay is and how many bytes are actually "in-flight". **gain** is a constant that controls the maximum change to the window in bytes per RTT.

The uTP specification recommends using 100 ms as **target_delay**, but not how to set **gain**. While LEDBAT RFC recommends **target_delay** 100 ms and **gain** less than 1*MSS bytes/RTT. The influence of these parameters have been previously researched, see section 2.8.2.

2.8.1 uTP Implementations

Many, but not all BitTorrent applications currently support the uTP protocol. The actual number of uTP implementations is much less and essentially consists of the libraries libutp[8], libtorrent[9] and libktorrent. Since these uTP implementations are available as open source they can more easily be studied and tested separately from the whole BitTorrent application protocol.

---

[9] BitTorrent client statistics vary quite a bit, this data is from the Tribler P2P research group at Delft University, available at [39].

[10] Vuze supports uTP through 'azutp' plug in which uses libutp on Windows/OSX.

Application	**uTP implementation**	**Popularity**[9]
uTorrent	libutp	48%
Vuze	libutp[10]	22%
BitTorrent	libutp	13%
Transmission	libutp	7%
BitComet	*unsupported*	1%
Deluge	libtorrent	n/a
qBitTorrent	libtorrent	n/a
Tribler	libtorrent	n/a
Ktorrent	libktorrent	n/a

Table 2.5: Bittorrent applications and their uTP implementations

2.8.2 Related work on uTP

There have been quite a few articles about uTP and many are naturally focused on the congestion control algorithm LEDBAT and the influence of the parameters **target_delay** and **gain**.

- “A hands-on assessment of transport protocols with lower than best effort priority”[40] - Performance comparison of different low priority congestion in TCP-LP, TCP-NICE and uTP-LEDBAT using ns2 simulations. Including inter and intra-protocol fairness, LEDBAT parameter sensitivity (target-delay and **gain**) and tuneability.
- “On the existence of optimal LEDBAT parameters”[41] - How LEDBAT parameters should be chosen to better achieve Lower-than-Best-Effort transfers that doesn’t impact competing TCP streams too much. Their conclusion is to use 5 ms target-delay and 10 as decrease gain (when delay_factor<0) and a low increase gain (when delay_factor>0) to limit growth to the TCP level. Compared to LEDBAT RFC, which says 100 ms **target_delay** and **gain** less than 1. Also notes that LEDBAT requires that queuing delay is possible, to be able to measure and react to it. They also found situations where LEDBAT could be more aggressive than TCP. Used ns2 simulations.
- “Assessing LEDBAT’s Delay Impact”[42] - That LEDBAT over time can grow queuing delay over the specified maximum target delay, because uTP over time will include its own induced delay in base_delay history. Verified against the libutp implementation. The simple solution that is suggested is to temporary stop the transfer minimum **target_delay** ms before updating base_delay. It is likely that

other implementations are affected since the issue originates in the protocol specification. Also shows that LEDBAT doesn't yield quickly enough for short TCP transfers.

2.9 Mininet

Mininet[5] is network emulator that uses existing Linux features to enable both virtual networks and light weight virtual machines. Mininet is implemented in Python and also offers a Python API to construct different environments and run tests. Using Python also makes it easy to implement automatic tests of many different test cases. Mininet also offers a way to run test manually after the network is constructed. [11]

The functionality it uses and provides is:

- Processes in network name spaces: Each process group can have different virtual network interfaces, routing tables and CPU usage limitations. This enables each process to be a simple ”virtual machine”. However, the hosts are not further isolated and all have same access to the host's file system.
- Traffic Control (TC), network emulation (Netem) and virtual network interfaces: To build virtual networks with different characteristics for example queuing disciplines, latency, bandwidth and packet loss.
- Software switches and routers. Mininet supports different software switches, and Mininet 2.2 supports virtual routers using Linux routing functionality.

Mininet was further developed as part of the Ph.d. thesis “Reproducible Network Research with High-fidelity Emulation”[43] which increased the accuracy of emulation with a set of tests to verify that the network parameters and corresponding invariants are true. The thesis also aims to use VM-containers in the cloud (Amazon EC2 virtual machines) to run network tests in self-contained environments. The idea is to publish network research in such VM-containers that can be distributed and then others can easily reproduce and continue the research.

Mininet has been used in the Stanford University course ”CS244: Advanced Topics in Networking” where students recreated results from network research papers, which originally didn't published a complete test environments. Many of these students' projects

---

[11] https://github.com/mininet/mininet/wiki/Introduction-to-Mininet#what

are now available, mostly in the form of Python scripts that uses Mininet and can run in cloud VMs[12].

The critical Linux functionality that Mininet uses, TC and NetEm, have previously been tested thoroughly, for example in [44] with results true to the input parameters, except when jitter in non-normal distributions. They have also been used in other papers such as [42] and [45].

Mininet uses the Linux Network Emulator, NetEm, to provide emulation of other network parameters:

Delay	One-way delay for all packets. The round-trip time is 2*delay.
Jitter	How much should delay change between different packets. Expressed as the standard deviation from a normal distribution.
Loss	How frequently are packets dropped, in percent.
Queue size	How many packets the send queue can hold. When full, the last is dropped. Test parameters are shown using the notation q=BDP*x (bits) which corresponds to a queue size of BDP*x/(8*1500.0) packets.

2.9.1 Python example

The basic usage of Mininet as shown below will create a network with two hosts, a switch and links with the emulated network properties: bandwidth, delay, no loss and limited queue size:

```
from mininet.net import Mininet
from mininet.link import TCLink
from mininet.node import OVSController
net = Mininet(link=TCLink, controller=OVSController)
h1 = net.addHost(’h1’)
h2 = net.addHost(’h2’)
s1 = net.addSwitch(’s1’)
c1 = net.addController(’c1’)
net.addLink(h1, s1, bw=10, delay="25ms", loss=0, max_queue_size=50)
net.addLink(h2, s1, bw=10, delay="25ms", loss=0, max_queue_size=50)
net.start()
h1.cmd("ifconfig")
h2.cmd("ping %s" % h1.IP())
net.stop()
```

---

[12] http://reproducingnetworkresearch.wordpress.com/about/

2.10 Brief overview of used tools

A few standard utilities were utilized and their functionality is briefly described here.

- Tcpdump - To record packets headers and/or data to packet capture files (PCAP-files), including time stamp when each packet was received.
- Ping/Hping3 - Utilities to send Internet Control Message Protocol (ICMP) echo packets and display round-trip-time, latency and packet loss.
- Iperf - Bandwidth measurement utility that can also function as a traffic generator.
- Nginx - Web server that supports HTTP and HTTPS.
- Curl - Command-line web client.
- Wireshark - Utility to graphically and interactively analyze network packets.
- Tshark - Command-line version of Wireshark to analyze and extract packet fields and statistics.
- Matplotlib - Python library to plot graphs.
- Numpy - Python library to perform statistical analysis.

Chapter 3

Methodology

The first part of this work naturally consisted of looking into available reliable UDP protocol and refreshing the knowledge about TCP from books like *Computer networking*[10] and *TCP/IP Illustrated, volume 1*[11]. SCTP, uTP and QUIC were chosen mostly due to their widespread usage on the Internet compared to other candidates such as Structured Stream Transport (SST)[46] or UDP-based Data Transfer (UDT)[47].

The next step was to read the protocols specifications and published articles to get an understanding of the protocols and current knowledge. This was also used to decide what kinds of tests that would be interesting to perform, such as verifying a specific feature or goal in a protocol or further analyzing a known problem.

The implementation of QUIC was tested because it's a new and relatively complex protocol which has not seen much public analysis yet. QUIC is also interesting because of its wide deployment and availability in the Chrome browser[6][48]. The implementations of uTP was selected because of the unique low priority congestion control, the wide usage[7] and that multiple implementations exists and can be compared. Previous papers that were found, such as [45] and [42], have only used libutp, which makes it interesting to test and compare another implementation. Time constraints limited the tests to the two most popular uTP implementations, libutp and libtorrent (see table 2.5). Because QUIC and uTP have widely different goals and properties it will be interesting to see whether Mininet can be useful to test both of them. SCTP implementations were not tested because of time limitations.

Since the goal was to test the actual protocol implementations in different scenarios, a network emulator was selected as the test method. This enables controlled experiments with actual implementations. Tests can be repeated while changing one parameter at the time. Tests can be repeated with constant parameters to see the variations of the results. This method was used to compare implementations in a wide variety of network situations and show the results in a few graphs, without having to test all combinations.

Emulators can be software or hardware based and runs in real-time and can usually interact with the real world. It can often use real implementations and applications. A downside is that emulators can be influenced by external factors such as current time, system performance and process scheduling. If a hard to reproduce bug is encountered, one cannot simple restart the test and expect the exact same sequence to occur again.

An issue when using network emulation is to know how accurate the emulation is compared to the selected parameters and "real-world" results. Even though Mininet has been verified before[43], the actual usage matter so verification tests of Mininet itself were performed as will be described in Chapter 4. The results have not been compared with tests in real networks, which is left as further work.

3.1 Alternative test methods

Often network simulators are used for analyzing network protocols. The main difference is that a simulator runs in a isolated environment with a virtual time which either can be faster or slower than real-time. This enables the simulation to run unaffected by external factors and results can be exactly repeated. The main downside is that real implementations or applications can't be used in a simulator; instead special simulation implementations must be used, which was not the goal of this thesis.

A downside of using network emulation is that the simplified models can't represent the full dynamics of network paths across the Internet. To do that, implementations can simply be tested on the Internet. This can be done at different scales from a few hosts, or using test beds for network research that can have hundreds of hosts or large scale tests involving millions of test hosts. The larger the scale the wider spectrum of network conditions can be tested. However, reproducing tests can also be a challenge on real networks since parameters can't be controlled and outside factors such as concurrent traffic will change over time.

Such testing depends largely on what resources are available. Test beds such as PlanetLab

EU[1] is only available for participating partners. Large scale tests can be performed by Google since they control both the server side and the Chrome browser[6].

The small scale available for this work would only have provided a limited set of different network properties, such as round-trip times, packet loss rates and bandwidths. This along with the problem of reproducing results made me dismiss this method.

3.2 Selection of tests

The tests were selected according to the thesis goals, and consist of two categories: performance and congestion tests.

The performance tests were selected to examine how the implementations perform in a variety of network conditions. When selecting between different implementations and TCP, one would want to know both performance benefits and weaknesses. The first test of QUIC was to repeat the Connectify test using the same network parameters. This was made to compare the previous result with a newer version of the QUIC implementation. To exercise QUIC's multiple streams, a test of transferring ten small files in parallel, a crude representation of a web page with resources. Also added were tests to see the influence of delay jitter, which is interesting because jitter can be common for wireless links. Additional test metrics were to compare the overhead, which is relevant especially since QUIC uses forward error correction to trade overhead for better performance. The performance tests of uTP uses the same selection of tests and parameters, because the same basic performance is also interesting to see here, and because the test system could be reused with few changes.

The congestions tests were selected to examine how the implementations react to concurrent traffic. This is relevant for any transport protocol to verify the fairness. These test have been performed for evaluation purposes, to know that the implementation function as intended and does not cause unintended problems. The two selected protocols have very different congestion goals so the tests were designed differently.

QUIC uses a new congestion control algorithm which is affecting its multiple streams. The tests were made to verify if the implementation are true to the specification. First by testing whether two QUIC connections are fair to two TCP connections, and secondly if a single connection with two streams in QUIC are fair to two TCP connections. While this could also be tested with many more concurrent connections, the result can be an

---

[1] `http://www.planet-lab.eu/`

indication whether further analysis is needed.

For uTP, the first congestion test was to see how the implementations react to occasional TCP connections, which is the main purpose of the whole protocol. It is interesting to see differences between the implementations for the same protocol. The second test was made to check if the growing queuing delay issue (see 2.8.2) can be replicated and to see if is still relevant for the implementations. Experiments with adding packet loss was also tried to see what effect it would have, because that was not tried in the previous work.

The implementations can be different in a number of areas, for example some features can be implemented without explicit requirement or support in the protocol. Other details are not specified which can lead to different implementations.

Chapter 4

Mininet

4.1 Rationale

Mininet was selected for several reasons.

The offered functionality seemed to coincide with the needs of the test cases. The ability to test real implementations in an emulated network environment with control over the different network properties. The envisioned network speed, number of links and processes were well within what Mininet can handle[43, p 64]. With a limited number of links which have bandwidths less than 100 Mbit/s and the total number of virtual machines much less than 100.

The ease of installation and that each 'host' doesn't require a separate "guest operating system" installation was also helpful. Because Mininet 'hosts' can access the same directory structure, there's no need to distribute binaries or test-files to the virtual hosts. All of this makes it an easy to use environment for test and development.

The ability to run tests on a laptop without requiring external hardware and to be able to easily move back and forth to a desktop computer was also helpful.

However, for testing non-application level such as different operating systems or kernel mode drivers, Mininet might not be as practical since all Mininet's hosts are running the same kernel.

4.2 Test system setup

The tests have been performed on a regular computer, running Ubuntu and utilizing Mininet. Appendix A contains the hardware specifications and exact software versions that have been used.

The tests of QUIC and uTP both requires similar test environments where different network properties and hosts where commands executed. Figure 4.2.1 shows the actual network layout that has been used in all of the tests. It is used to model the *dumbbell* layout where a number of hosts talk to other via a limited link in between. The extra switch *switch1* was needed to be able to capture packets after being subject to the restricted link.

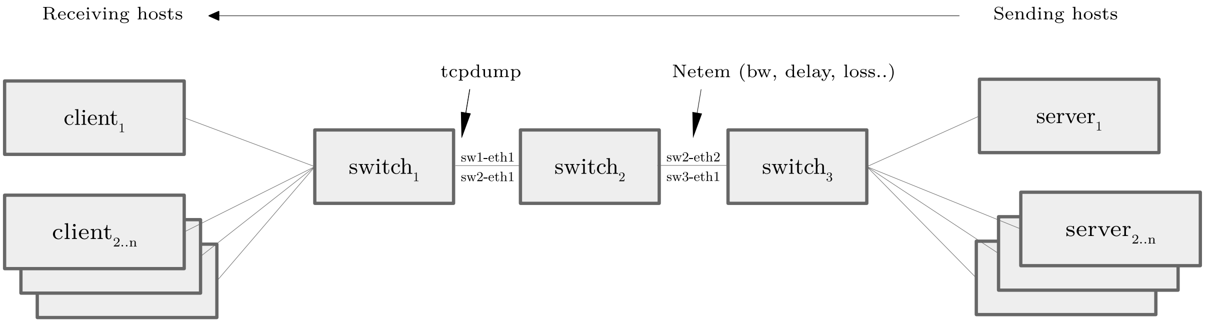


Figure 4.2.1: Utilized network layout in Mininet

All network links can have different properties, but in this thesis only the link between $switch_2$ and $switch_3$ have the interesting properties. All other links have unlimited bandwidths and packet-queues, and uses no delay, jitter or packet loss.

The hosts called $client_1$ and $server_1$ will be used for the main protocol under test, while $client_{2..n}$ and $server_{2..n}$ can be used to run other network traffic simultaneously, *cross-traffic*, to study congestion and fairness properties.

Two different test systems were been developed to run test cases. They are both very similar, and use simple data structures to define a number of test-cases, which describes network properties, the commands to run and how to capture and extract results.

mininet-performance.py is used to run performance tests to compare how the protocols handle different network environments. Each test case consists of a set of network parameters and the varying parameter's test interval. The protocols are then tested repeatedly while different results are captured, such as completion time and the number of transmitted packets. These results are later plotted to show how the varying parameter (on X-axis) yields different results (Y-axises). Each set of network parameters is also repeated a number of times to capture the variance of the result, which is shown using an error-bar with minimum, median and maximum in the plots.

mininet-congestion.py is used to run congestion and fairness tests to show how the protocols utilize bandwidth when multiple protocols are used simultaneously. Each test case describes the network parameters and a schedule when the commands should run. Tcpdump is used to capture packet headers into a PCAP file for each test case. Then Tshark is used to extract the number of bytes transmitted per time slice for each connection and plotted. Similar to the "IO Graph" functionality in Wireshark[1] but done automatically from a script. Can optionally include ping RTT measurements and Jain's Fairness Index calculations.

4.3 Test system verification

Verification that a test system works as intended is of great importance and this have previously been done extensively on Netem and Mininet, for example in [43] and [44].

Since the combination of used hardware, operating system, software versions and test scripts also will influence the result; a few initial test verifications have been performed. These tests were also helpful in trying to better understand how the test system behaves and the influence of different parameters.

4.3.1 Packet delay and jitter

Packet delay is specified in milliseconds as the minimum one-way delay. The total delay can be larger depending on current queuing delays. Jitter adds randomness to the delay of each packet and is specified as the standard deviation in milliseconds from the normal distribution. Netem also supports other random distributions, but these have not been used.

To verify delay, ping packets were sent from client1 to server1 and the RTT time was captured with Tcpdump at switch1[2]. This round trip time should be close to $2 * delay$

---

[1] Menu option: Statistics →IO Graph

[2] To get higher RTT precision than normal ping output

since the delay is applied in both directions.

Target delay	Average RTT	Standard error	Min	Max	Sent packets
0 ms	0.054	0.045	0.028	0.490	100
10 ms	20.091	0.011	20.066	20.125	100
25 ms	50.093	0.012	50.069	50.123	100
50 ms	100.095	0.015	100.066	100.137	100

Figure 4.3.1: Validation of delay

To verify jitter, ping packets were sent from server1 to client1 at specific intervals (20 ms) and their inter-arrival times were recorded at switch1 with Tcpdump and extracted with Tshark. The average inter-arrival time should be close to the sending interval and standard deviation of inter-arrival times should be close to the specified jitter. The arrival times were also plotted using a histogram to verify the normal distribution.

Target jitter	Arrival standard error	Arrival average	Min	Max	Sent packets
0.0 ms	0.019	20.064	19.812	20.120	450
1.0 ms	0.807	20.075	18.191	22.003	450
2.0 ms	1.607	20.077	16.583	23.981	450
3.0 ms	2.436	20.079	14.677	25.843	450
4.0 ms	3.078	20.064	12.706	27.028	450
5.0 ms	3.970	20.067	10.799	29.445	450

Figure 4.3.2: Validation of delay jitter

The actual jitter is thus a little lower than the specified value, which is a known issue in Netem[44].

The specified jitter values in this thesis are the actual inputs, not adjusted inputs or resulting jitter.

4.3.2 Packet loss

Packet loss is specified as the probability p that a packet is loss. Netem also supports different models of packet loss correlation (e.g. Gilbert-Elliot or Bernoulli) but these have not been used.

To verify this property, x ping packets are sent from server1 to client1 and the received packets are recorded on switch1 using Tcpdump, where close to $x - x * p$ packets should be seen.

Target loss	**Actual loss**	**Receive count**	**Sent packets**
0.0 %	0.00 %	1500	1500
0.1 %	0.13 %	1498	1500
0.5 %	0.46 %	1493	1500
1.0 %	1.20 %	1482	1500
2.5 %	2.13 %	1468	1500
5.0 %	4.13 %	1438	1500

Figure 4.3.3: Validation of packet loss

The actual loss rate is close to the target loss, but for higher loss rates accuracy it is not very exact.

4.3.3 Queue length

Tested by burst sending 10*queue_size number of packets from s1 and counting the number of packets that arrives at c1. When the packet queue is full, additional packets should have been dropped. This requires the total sending time to be less than the one-way delay.

Different bandwidths and packet-sizes were used with 100 ms RTT to verify this property.

Bandwidth	**Received packets**	**Queue length**	**Sent packets**
1	50	50	500
5	50	50	500
10	50	50	500
25	50	50	500
50	50	50	500
100	50	50	500

Figure 4.3.4: Validation of queue length

4.3.4 Bandwidth and congestion

Verified by running Iperf between $client_n$ and $server_n$ for $n \in (1, 2, 3)$. Tested multiple available bandwidths and verified that both TCP-Reno and TCP-Cubic could utilize it.

This test was also interesting to get an idea how regular TCP fairness would look like. While the individual stream's share varied quite a lot between different runs the feeling was that TCP-Reno overall resulted in better fairness both visually and by Jain's Fairness index. However, not enough runs were made to analyze the fairness values statistically.

See figures 4.3.5, 4.3.6 and 4.3.7 for example results of TCP-Reno, which utilizes the available bandwidth with good fairness numbers (all above 0.92). The last chart with the 50 Mbit/s bandwidth limit shows that the exponential slow-start ends before 5 Mbit/s per stream before continuing with a linear increase, which indicates that TCP's slow-start threshold could be set higher.

TCP-Cubic results are shown in 4.3.8, 4.3.9 and 4.3.10. Where the first two also show good bandwidth utilization and fairness scores (both above 0.95), while the last chart with 50 Mbit/s shows that the bandwidth can't continuously be fully utilized and the fairness score is quite bad - "tcp3" stream hardly use any bandwidth. Repeating this would be necessary to see if this is a one-off result or not. But it is reasonable to suspect that the queue length needs to be larger to get full bandwidth utilization.

4.4 Issues and solutions

Mininet 2.1 could only use switches as intermediate nodes. This makes the network require Address Resolution Protocol (ARP) requests and responses, which can introduce unwanted effects with testing network congestion and network delay. When this was realized, there was a convenient option to be found in Mininet[3] to add static ARP entries and avoid this issue.

To capture the result of packets after passing the limited network link there was a need to add switch1. If switch1 was removed and Tcpdump used on switch2 (interface sw2-eth2) the packets from switch3 would be captured before being subject to network parameters. If Tcpdump could listen on the specific interfaces between clients and the switch, the extra switch would not be necessary.

---

[3] autoStaticArp=True

The results for 0%, 0.1% and 0.5% packet loss were initially the same, which was solved by adding support for decimals in Mininet when constructing Netem parameters for loss.

When looking at pcap files, it became apparent that TCP used very large IP packets (>10 KB). Even though the emulated links had the proper MTU set (1500) and the MSS Option field was also correct (1460). The cause was eventually found and turned out to be the ”Offload Engine” in the Linux kernel. Even virtual network adapters could apparently be subject to ”hardware” acceleration. This problem was solved by using *ethtool* to disable ”TCP segmentation offload” for all virtual adapters. All other types of offloading were also disabled to avoid similar problems elsewhere. This is something that is not fixed or documented in Mininet and it doesn’t look like anyone has noticed this. Such large packets from TCP made it look like the other protocols use very many packets in comparison.

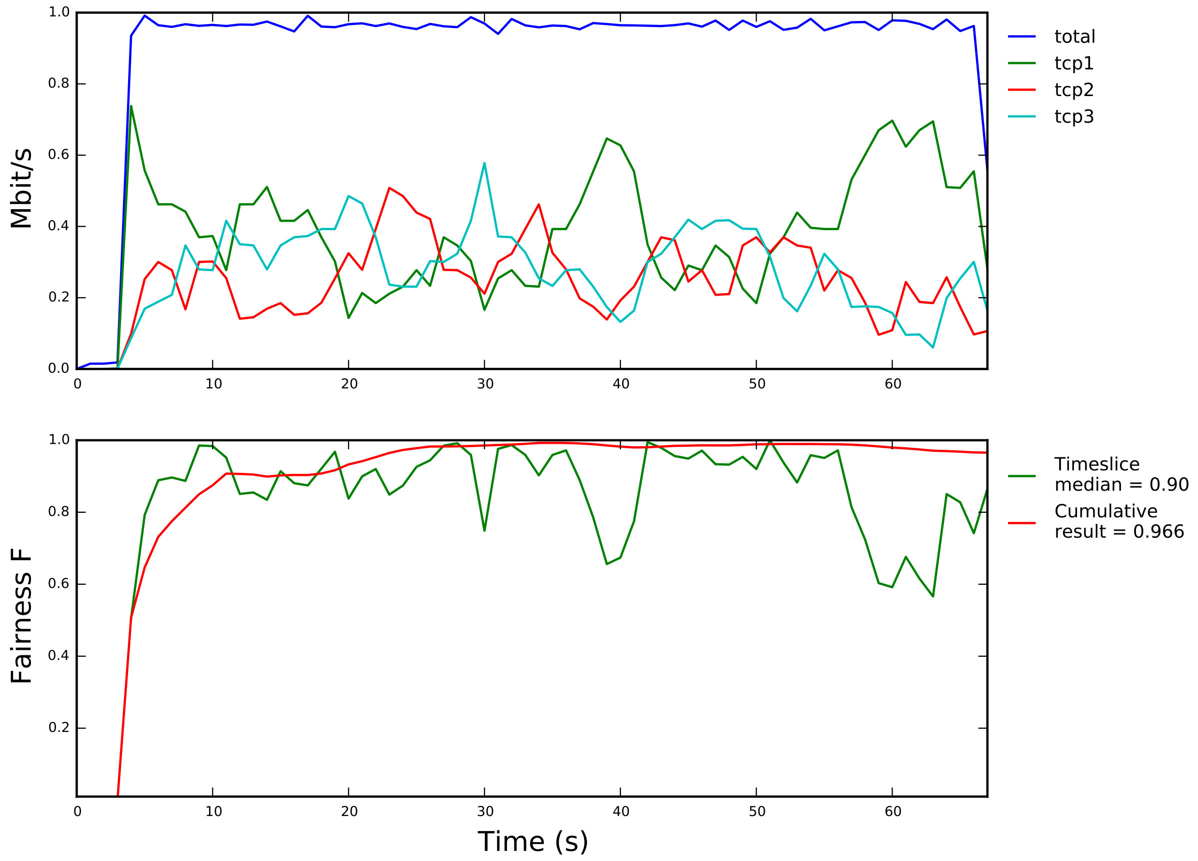


*Figure 4.3.5: Bandwidth utilization for 3 x TCP-Reno, started at t=3, rtt=100 ms, bw=1 Mbit/s, q=BDP*1*

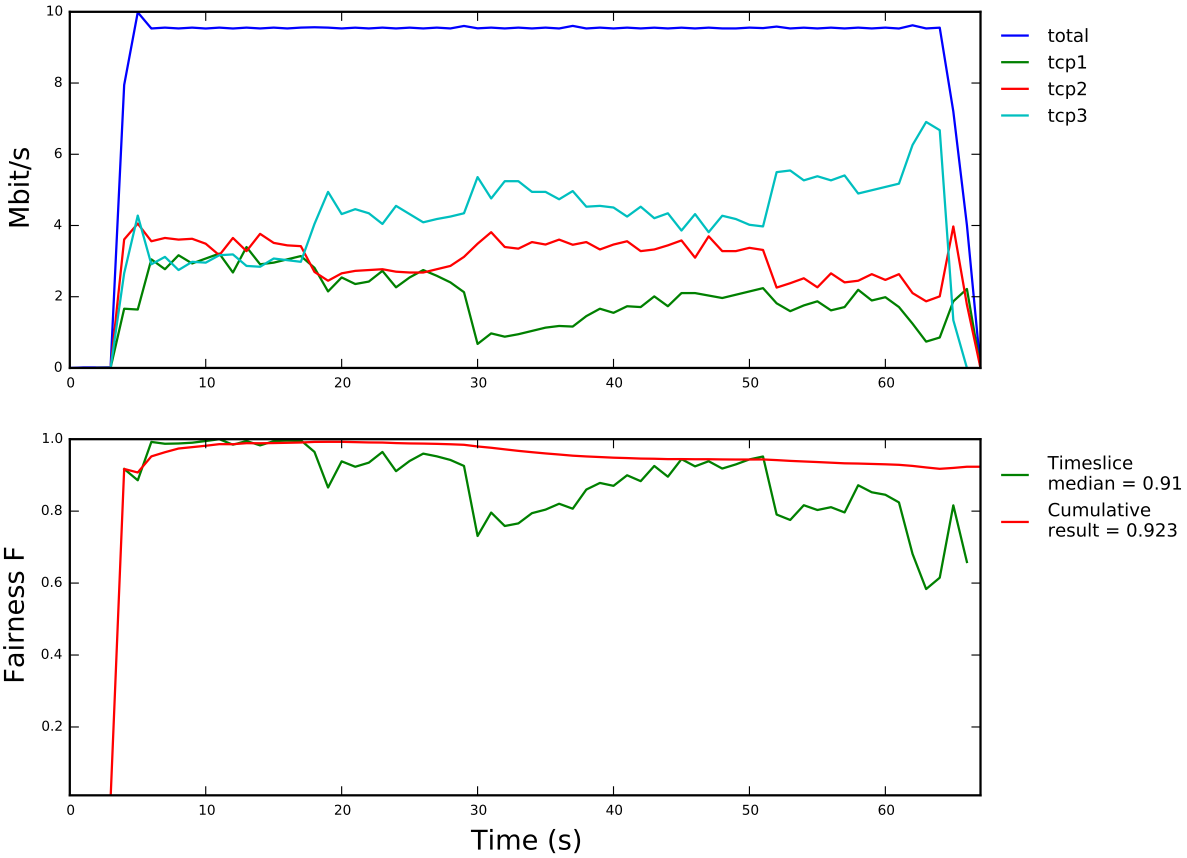


*Figure 4.3.6: Bandwidth utilization for 3 x TCP-Reno, started at t=3, rtt=100 ms, bw=10 Mbit/s, q=BDP*1*

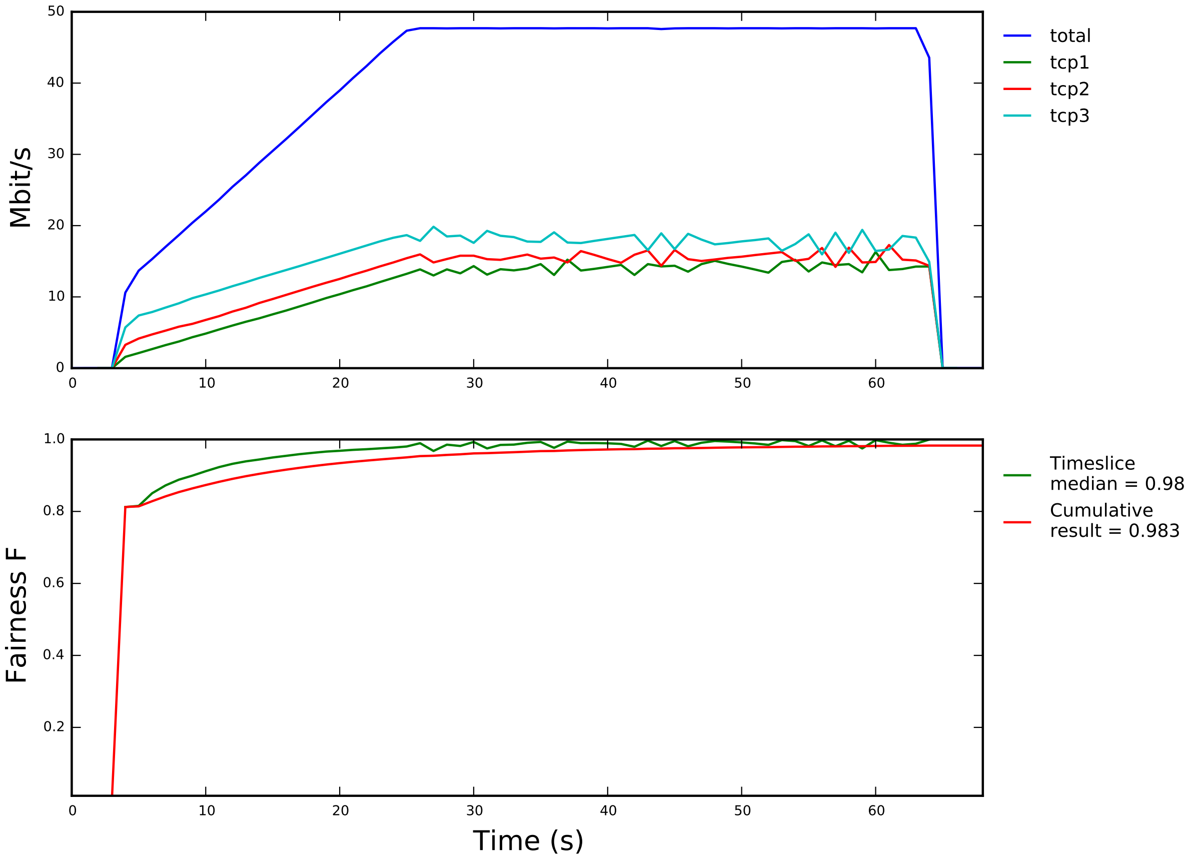


*Figure 4.3.7: Bandwidth utilization for 3 x TCP-Reno, started at t=3, rtt=100 ms, bw=50 Mbit/s, q=BDP*1*

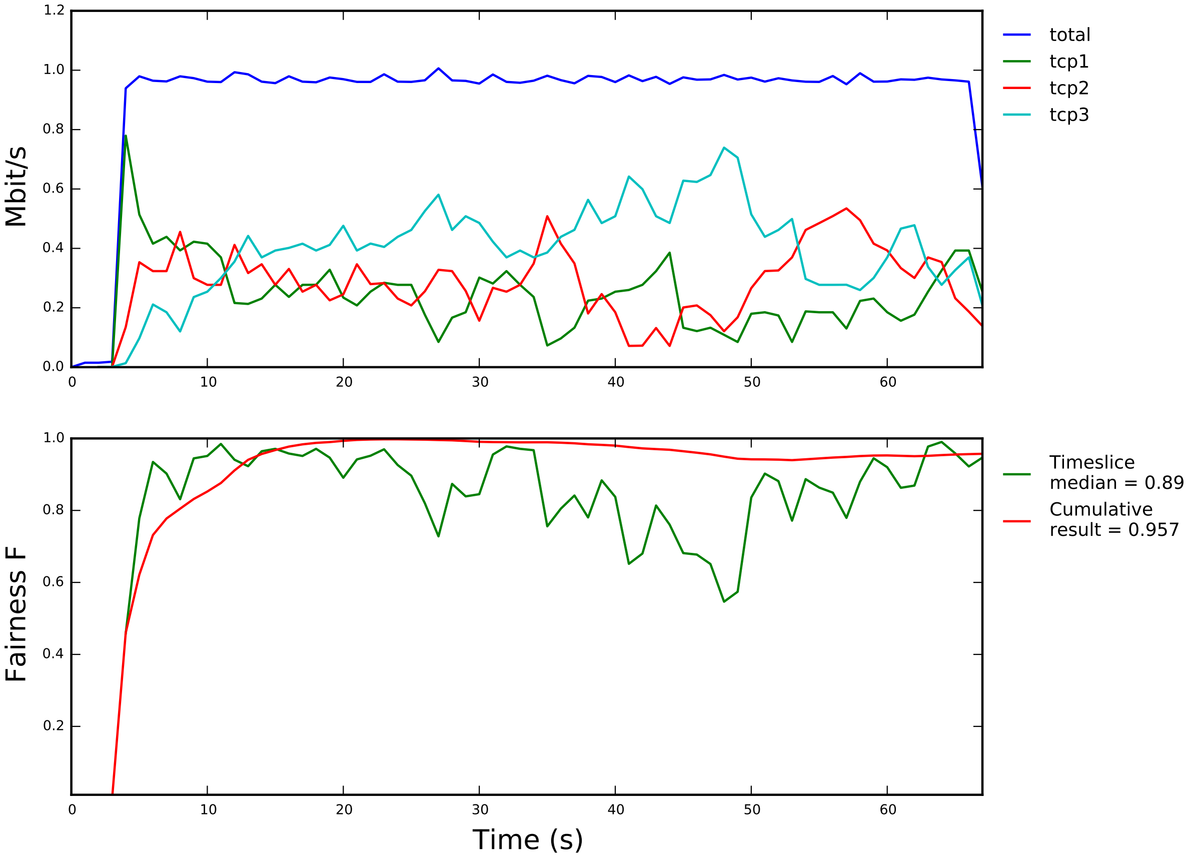


*Figure 4.3.8: Bandwidth utilization for 3 x TCP-Cubic, started at t=3, rtt=100 ms, bw=1 Mbit/s, q=BDP*1*

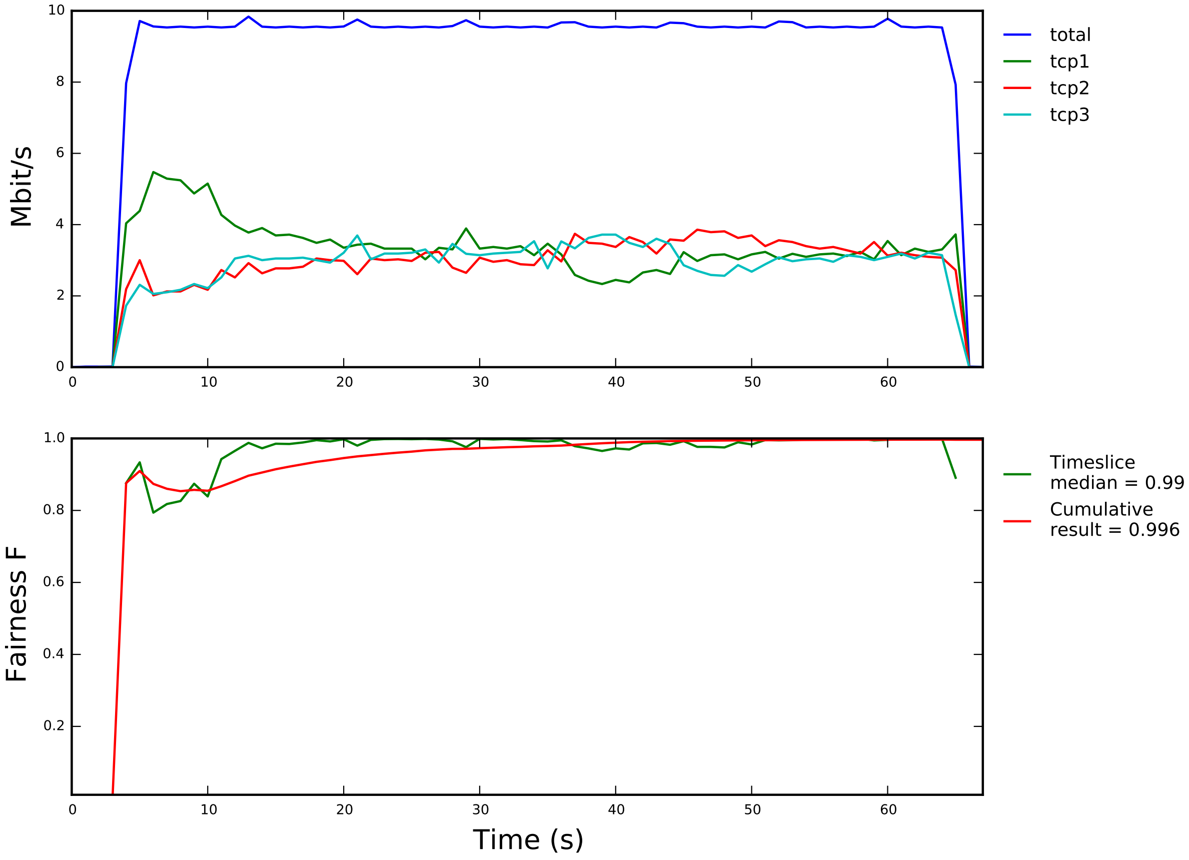


*Figure 4.3.9: Bandwidth utilization for 3 x TCP-Cubic, started at t=3, rtt=100 ms, bw=10 Mbit/s, q=BDP*1*

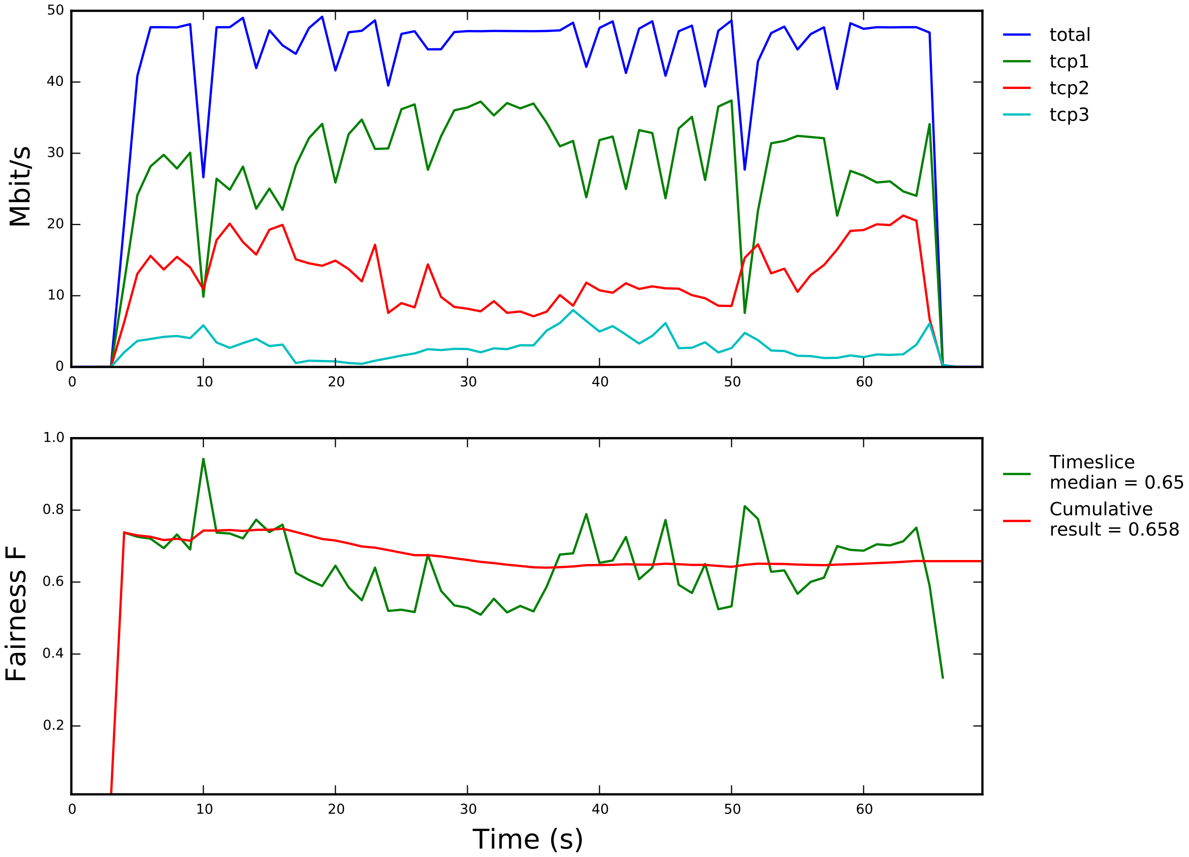


*Figure 4.3.10: Bandwidth utilization for 3 x TCP-Cubic, started at t=3, rtt=100 ms, bw=50 Mbit/s, q=BDP*1*

Chapter 5

Google QUIC

5.1 Test setup

QUIC has been tested using Mininet to measure performance and study congestion handling. The performance tests include comparisons with the TLS protocol, which QUIC intends to replace, and also TCP as an additional reference. TLS and TCP were performed using the web server *Nginx* and the web client *Curl*. The congestion tests used *Iperf* for generating competing TCP streams that used the TCP-Cubic congestion control.

To test QUIC the utilities *test-client* and *test-server* were used. They could be used from a scripting environment to run the different tests automatically.

First, the Chromium source code was downloaded and the utilities compiled. The functionality was verified and a few minor issues that prevented testing were resolved.

- Output file - By default the test programs could only show output on standard output, and there was a problem of using file redirects ('>') with Mininet. An option to save the output to a file was added.
- Output NULLs - The whole response is now saved in the output file, not stopping at NULL characters as the original did.
- Output headers - An option was added to hide the HTTP header in the output file. This was to make it easier to verify that the resulting files were transferred

correctly by comparing file checksums.

- Listen IP - The original test_server could only accept connection from localhost, preventing the virtual hosts with other IP addresses in Mininet to connect. This patch is the same as the one Connectify used.

A few words of caution of the QUIC test tools. Since they doesn't save state such as session keys between different runs, the improvements in connection establishment (0-RTT) has not been utilized in these tests. Also included in the timing measurement is the connection close sequence, which doesn't really affect page load time in a browser.

5.2 Connectify performance comparison test

The first test was performed according to the same network parameters used in the Connectify test - Talking Google's QUIC for a test drive (see 2.7.2). Mainly to see how the newer QUIC version (2014-09-01) performed compared to Connectify's version (2013-11-01), and partly to see if HTTP results was similar even though the test system was different. The main difference was the usage of Mininet instead of an external Linux router with Netem.

In addition to HTTPS comparison, another improvement is that each test is repeated 5 times and shown in the graphs as error bars with the minimum, median and maximum values. This is used to show variations in the results.

5.2.1 Connectify performance comparison results

The results for HTTP are almost identical to Connectify's in all three cases, which shows that the test systems will produce very similar results.

The bandwidth test in figure 5.2.1 shows that QUIC no longer has the 8 Mbit/s goodput limit, and can follow HTTPS until available bandwidth is 30 Mbit/s, after which goodput don't increase linearly with available bandwidth. QUIC ends at 36 Mbit/s and HTTPS at 56 Mbit/s.

The reason for QUIC not performing as well in the higher bandwidths can be understood using Wireshark's "IO Graph" for PCAPs from the 70 Mbit/s test. The 10 MB file

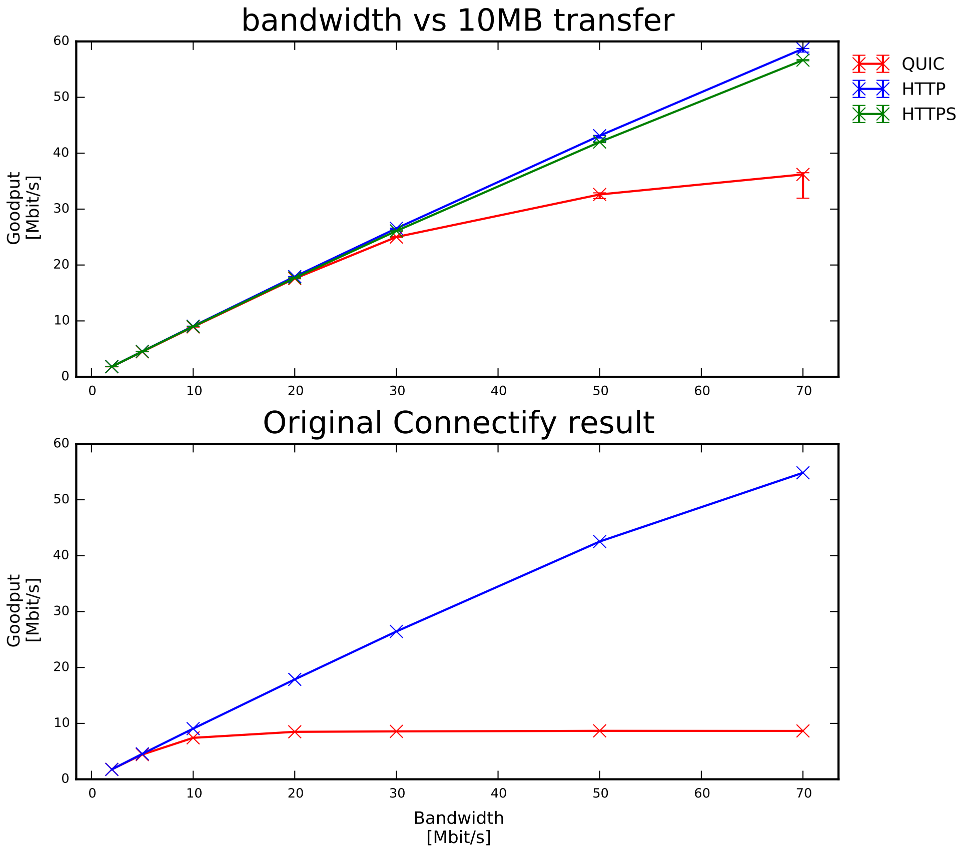


Figure 5.2.1: Bandwidth test 2-70 Mbit/s, 20 ms RTT and 0% loss

completes very quickly and QUIC doesn't increase it's transfer speed as fast as TCP. A larger test file would have shown more equal QUIC goodput.

The latency test result in figure 5.2.2 shows that QUIC have improved it's performance for low latencies and are now equal to HTTP. However, latencies above 200 ms have a large impact of QUIC performance.

The packet loss in figure 5.2.3 shows how QUIC forward error correction will improve goodput when there's packet loss. QUIC is almost unaffected by 0.1% loss and for higher loss it has twice the speed compared to HTTPS. The overhead cost of the parity packets will be shown later.

However, even if QUIC could recover from a packet loss the congestion window has to be reduced. But is QUIC's reduction enough for "relative fairness"?

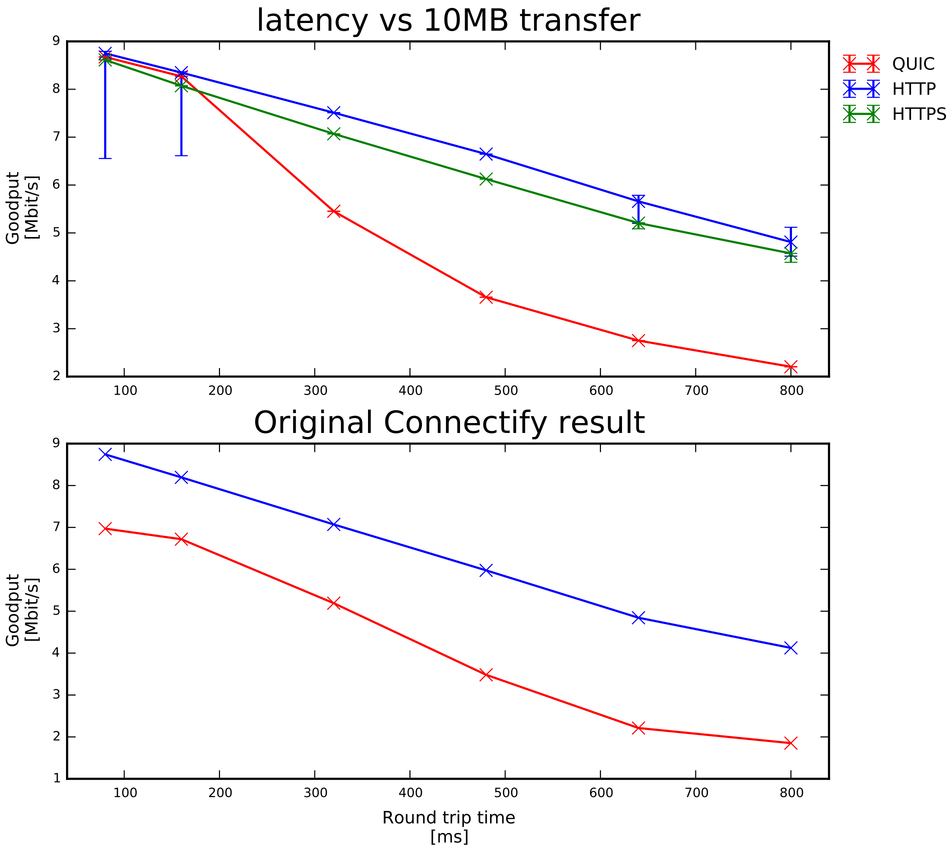


Figure 5.2.2: Latency test 80-800 ms RTT, 10 Mbit/s bandwidth and 0% loss

5.3 New performance tests

Performance tests have been performed in another a scenario that transfers multiple files on a single connection to test the stream multiplexing feature in QUIC. This can be performed using the standard QUIC test tools. The TCP comparison uses curl with sequential requests on the same connections. Ideally, the comparison should have been done with multiple parallel TCP connections.

A web page with additional resources come in an many different shapes, which makes it difficult to choose the number of files and file sizes to test. HTTP is also often used to load dynamic content through XMLHttpRequest of widely different sizes. The tests here were made using 10 files of 100 KB each, giving a total size of 1000 KB as a crude representation of a web page.

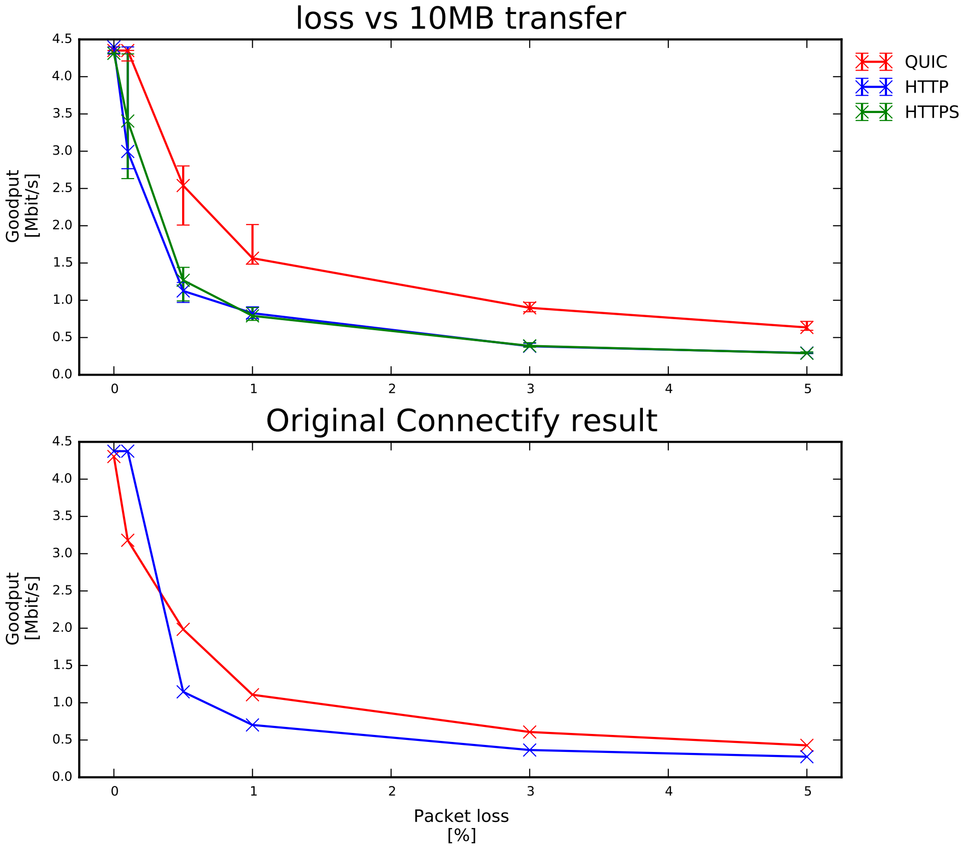


Figure 5.2.3: Packet loss test 0-5%, 5 Mbit/s bandwidth and 160 ms RTT

The network parameters are detailed below each graph and are based on the Connectify parameters with a few adjustments. The latency case is now using 10-500 ms RTT instead of 80-800 ms because that felt more relevant. The loss case also uses a lower RTT 100 ms instead of 160 ms for the same reason.

A new test was added to study the influence of delay jitter on goodput, which can be important to know for mobile applications.

To see the overhead of forward error correction, the plots also shows how much extra data (sent-bytes) and how many packets (sent-packets) the server transmitted.

5.3.1 New performance tests results

The bandwidth test in figure 5.3.1 shows how QUIC continues to grow with available bandwidth, while HTTP and HTTPS is limited at 10 Mbit/s goodput even when available bandwidth increases.

Because HTTP and HTTPS could only request one file at the time in this test, each file will cause a delay of at least one RTT. For these small file sizes that will have a large impact on transfer rate.

The overhead of QUIC's parity packets is about 1.6 percentage points higher than HTTP when the bandwidth is above 5 Mbit/s.

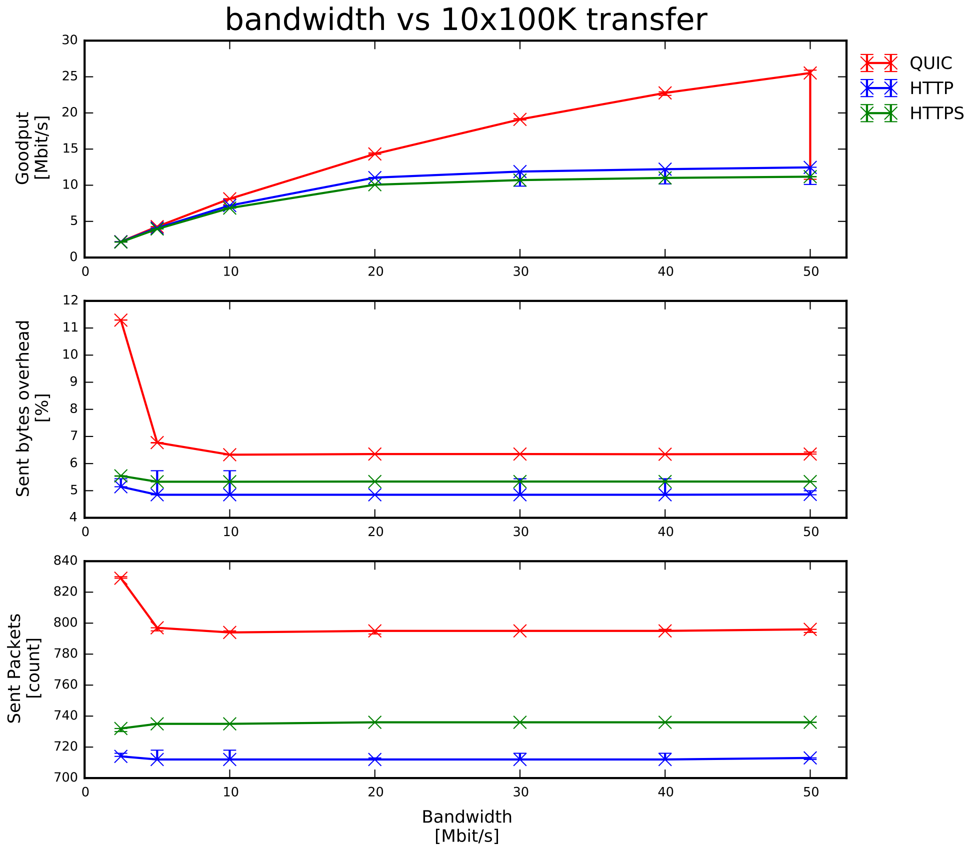


Figure 5.3.1: Bandwidth test for 10 files, 2-50 Mbit/s, 20 ms RTT, 0% loss

The latency test in figure 5.3.2 shows that QUIC is still affected by the increase of RTT, but less than TCP. If multiple parallel TCP requests had been used, the latency dependency would have been lower.

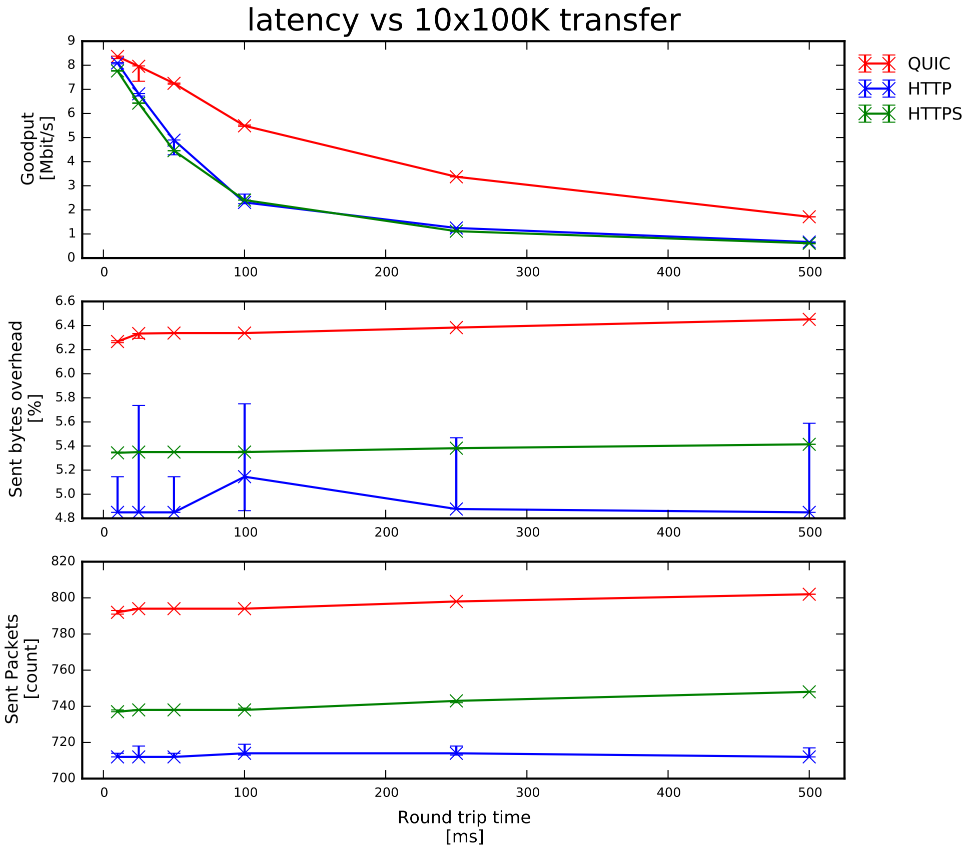


Figure 5.3.2: Latency test for 10 files, 10 Mbit/s, 10-500 ms RTT, 0% loss

The loss test in figure 5.3.3 shows how QUIC clearly outperforms when there's packet loss. QUIC speed is almost unaffected by 0.5% loss or less. The larger variation in QUIC goodput compared to HTTPS, could be explained that with FEC, QUIC can only sometimes recover for losses depending on which packets are lost.

Also interesting here are the overhead numbers, in bytes and packet count. It looks like QUIC's parity uses around 1.6 percentage points more bytes than HTTP, or 1 parity packet every 62 data packets. For all protocols the byte counts increase is closely related to the loss, i.e. increase of 1% loss requires 1% more bytes to be transmitted. QUIC is only designed to handle low loss, also QUIC increases linearly with greater loss.

The impact of loss for packet count is also almost linear, but it is much less visible because of the scale used in the plot.

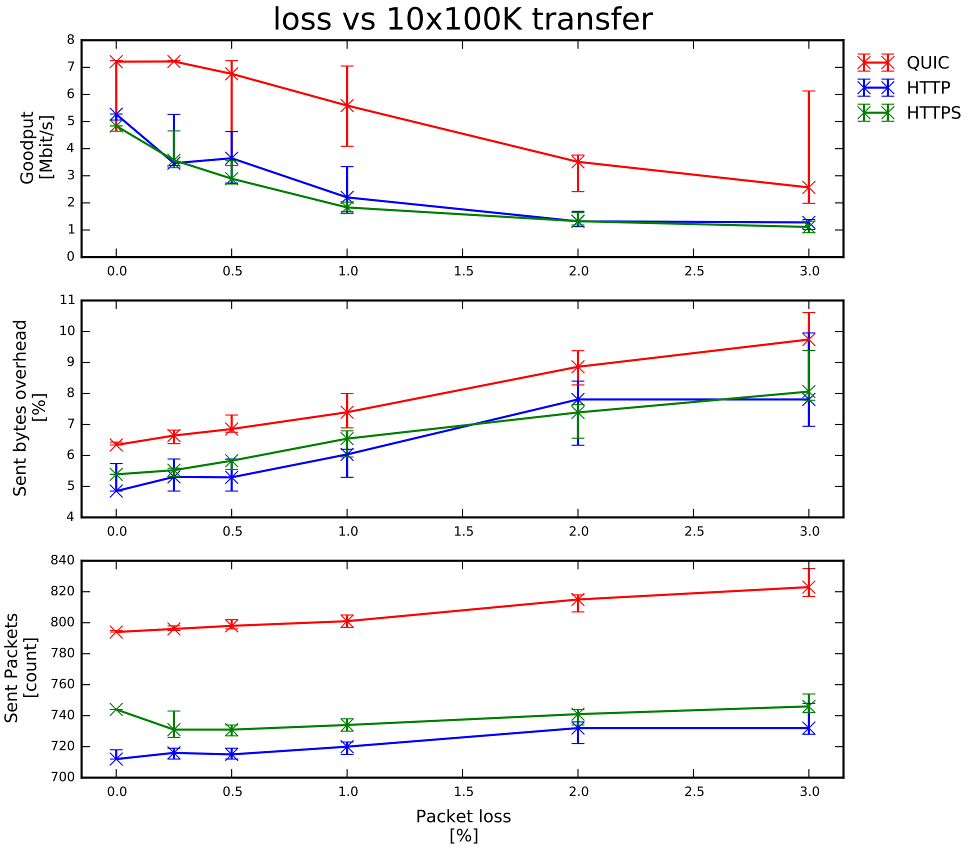


Figure 5.3.3: Loss test for 10 files, 10 Mbit/s, 100 ms RTT, 0-5% loss

The jitter test in figure 5.3.4 show that QUIC is very affected by even small amounts of jitter in packet delays. The effect of normal distributed jitter with 0.5 ms standard deviation to the 25 ms one-way delay (50 ms RTT) cuts the transfer rate from 7 to 2 Mbit/s.

Similar to uTP, it seems that QUIC also measures the packet's inter-packet arrival times and adjust the sending rate.

The result of 10 ms jitter seems to trigger a bug or corner case in the QUIC implementation because the number of packet transmission goes through the roof.

The impact on jitter for HTTP and HTTPS is almost none.

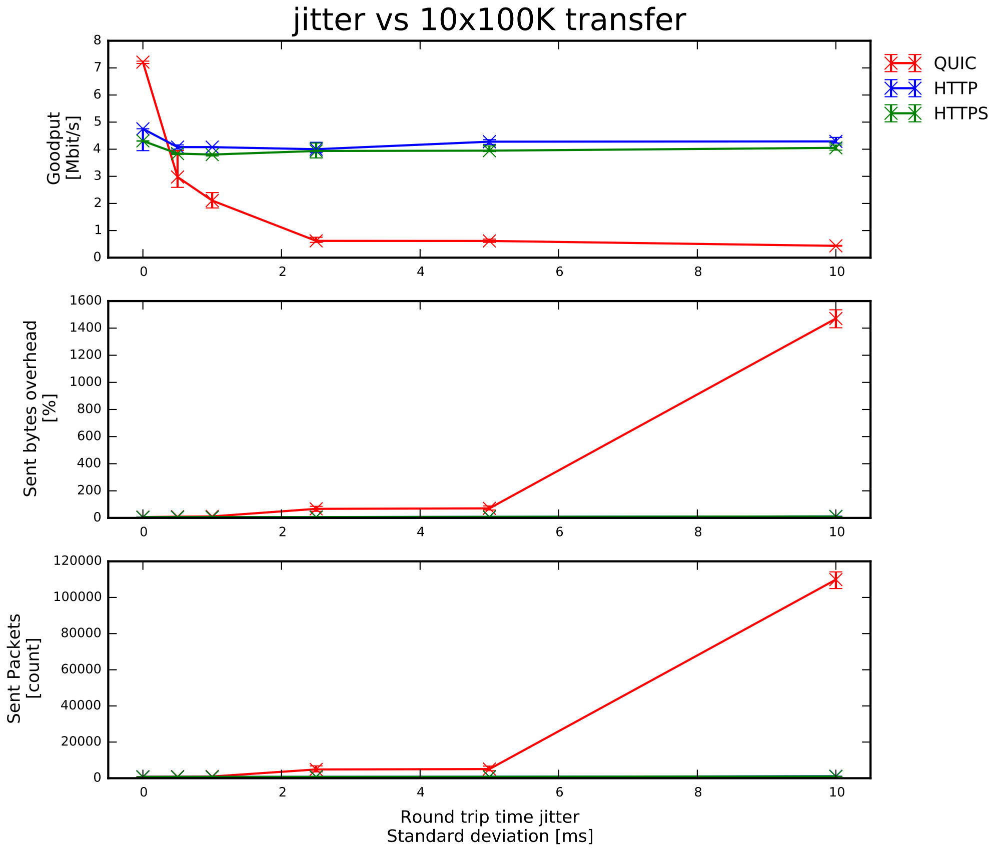


Figure 5.3.4: Delay jitter test for 10 files, 10 Mbit/s, 50 ms RTT, 0% loss, 0-10 ms std jitter

5.4 Congestion tests

QUIC's basic fairness handling against TCP was tested by running multiple large transfers by TCP-Cubic using Iperf and QUIC using test_client and test_server.

First, two separate QUIC connections and two TCP connections, all started at the same time. The bandwidth utilization should be shared roughly equally, and the overall Jain's Fairness Index should be close to 1. The test was repeated 30 times and the median fairness calculated.

Then, one QUIC connection that has two streams and two TCP connections, all started at the same time. The goal of QUIC is that it's two streams should use the same bandwidth as two separate TCP-Cubic streams. Jain's Fairness was not calculated since only the combined bandwidths of the two QUIC streams were available. This was also repeated 30 times and the number of runs where QUIC used more total bandwidth than the two TCP was counted.

5.4.1 Congestion tests results

A single result from the first test can be seen in 5.4.1, where the two QUIC streams used 15.2 Mb and two TCP used 8.9 Mb, giving a fairness of 0.92. When repeating this test 30 times, QUIC used more bandwidth than TCP in all of them. The median fairness was 0.84 which is even less than the test pictured.

From the second test, a single result in shown in 5.4.2, where a QUIC connection's two streams uses 13.5 Mb and the two TCP uses 10.6 Mb. When repeating this test 30 times, in 60% of the cases QUIC used more bandwidth than TCP.

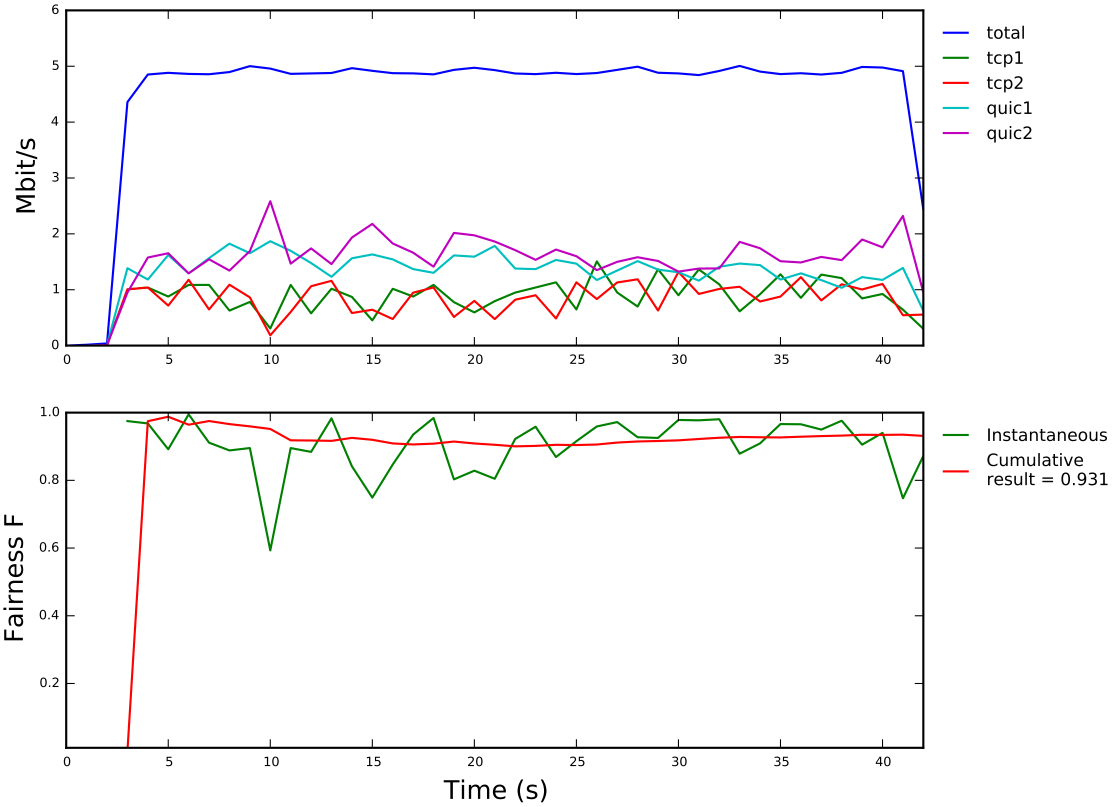


*Figure 5.4.1: Congestion test, 2 x TCP and 2 x QUIC, bw=5 Mbit/s, q=BDP*5, RTT=100 ms*

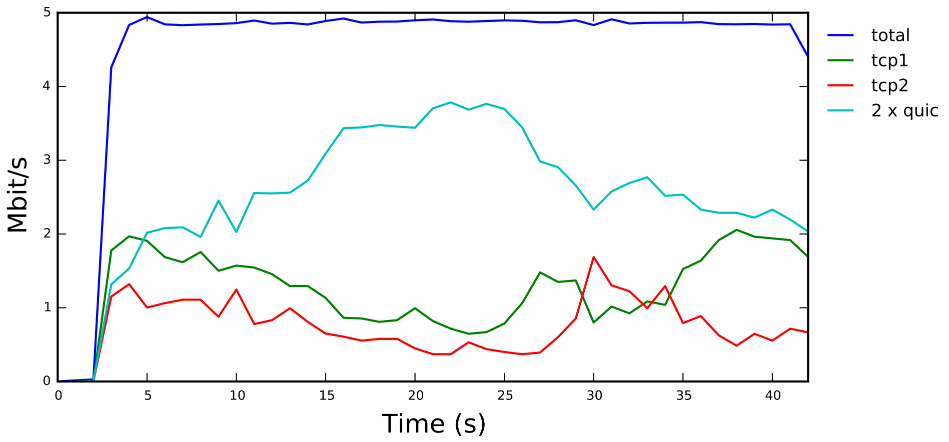


*Figure 5.4.2: Congestion test, 2 x TCP and 1 QUIC with 2 streams, bw=5 Mbit/s, q=BDP*5, RTT=100 ms*

5.5 Conclusions for QUIC

One of the problems with the Connectify tests was that the used version of the QUIC implementation didn't have FEC enabled and that CWND had been temporarily limited. Using a newer version we can now see that both FEC is working and that CWND no longer have that limitation.

The performance in low delay without packet loss is very similar to HTTPS, except for higher bandwidth which is mostly due to the slower CWND growth rate. The tests confirm that QUIC does work well when there's a small probability of packet loss, which is what QUIC is designed for. To cover larger packet loss would result in even more overhead, but packet loss must also affect the congestion control even if the packets can be recovered. The performance in high latencies could need some improvements to match HTTPS results, although having more than 300 ms latency is less common today.

The performance in delay jitter probably does need a closer look. When used over a wireless network that often have naturally occurring jitter, such as a LTE network, QUIC should use another congestion algorithm that doesn't use packet timing as an indication of network congestion. TCP is unaffected by jitter since it typically only uses packet loss as the congestion indicator. There's already support for negotiating which congestion control to be used in QUIC, so a mobile app could use the current network technology to decide. QUIC could also fall back automatically by detecting when the jitter is above a certain threshold and then renegotiate the congestion control.

Looking at the transfer of multiple files, it's clear that QUIC is faster than sequential HTTPS requests. However, since web sites will use multiple domains to avoid connection limitations, the test should have used multiple parallel HTTPS connections to compare.

The tests in this thesis also show that the variations in the results, which can be caused both by the protocols and test system, for most parts have been quite low. One exception is the packet loss case with its random selection of lost packets, which QUIC can either repair directly or both need to resend and lower the CWND, causing variations in QUIC's performance.

The tests of QUIC's congestion handling show that the case of one QUIC connection with two streams seems to use just a little more than two TCP connections, and could still be considered fair. However, the case with two separate QUIC connections and two TCP connections strongly suggests that QUIC is not completely fair. More analysis is needed to understand exactly when and why, to properly adjust the algorithm or parameters to be more TCP-friendly.

Chapter 6

uTorrent Transport Protocol (uTP)

The basic performance characteristics and congestion handling in uTP has been tested using similar tests as with QUIC.

Additionally, tests were made to see if the growing queuing-delay issue identified in “Assessing LEDBAT’s Delay Impact”[42] is still present in libutp and if libtorrent is also affected. Experiments were also made to see how a low rate of packet loss would affect this issue.

6.1 Test setup

In order to test the uTP implementations, simple file transfer programs were developed for each implementation. These can either wait for a connection and then send a file or connect to another host and receive a file.

libutp already had a similar utility called ’ucat’ that could easily be adapted to be used as the file transfer program.

libtorrent was a greater challenge since there were no existing stand-alone utility for it’s uTP implementation. This work consisted of figuring out how to use uTP separately and implementing a stand-alone test program.

6.2 Performance tests

The performance tests are similar to the QUIC performance tests in section 5.3, using same network parameters and a 10 MB test file instead. The same measurement, goodput, or effective bandwidth is also used. Each test is repeated 5 times and shown in the graphs as error bars with the minimum, median and maximum values.

The TCP comparison used the same setup of Nginx and Curl.

6.2.1 Performance tests results

The bandwidth result is shown in 6.2.1 where TCP can utilize the available bandwidth linearly, closely followed by libtorrent with gap to libutp which isn't following linearly.

The differences between libutp and libtorrent depends almost entirely of the initial CWND growth, slow-start, where libtorrent's growth is faster than libutp. For the high bandwidths, this makes a big difference since the total transfer time is so short for a 10 MB file.

The initial growth will be shown later in the congestion test.

The overhead graphs show how libutp uses a higher packet count than libtorrent. This can be explained by libutp having a 1430 byte limit on IP packet size. libtorrent have built in MTU probing and can use 1500 byte IP packets.

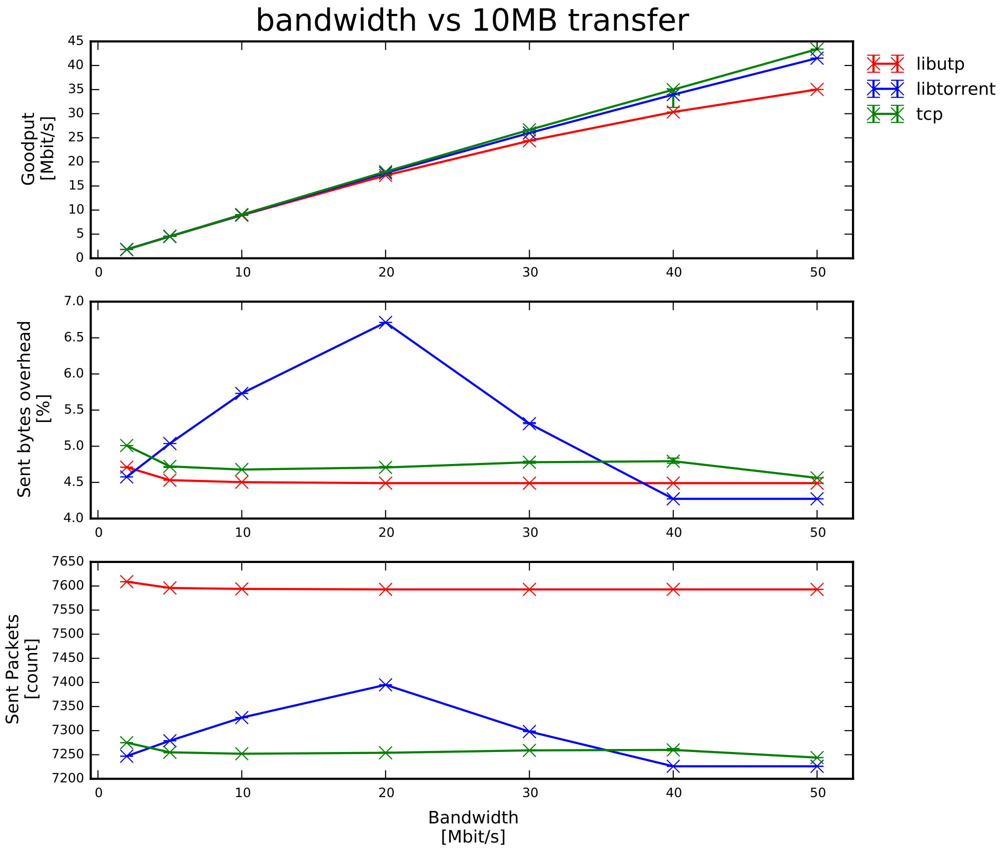


Figure 6.2.1: Bandwidth test, 2-50 Mbit/s, 20 ms RTT, 0% loss

The latency test in figure 6.2.2 shows that libtorrent handles the increase quite well, while libutp loses much performance as latency increases. This can also be explained by the slower CWND growth, which is more affected by the latency increase.

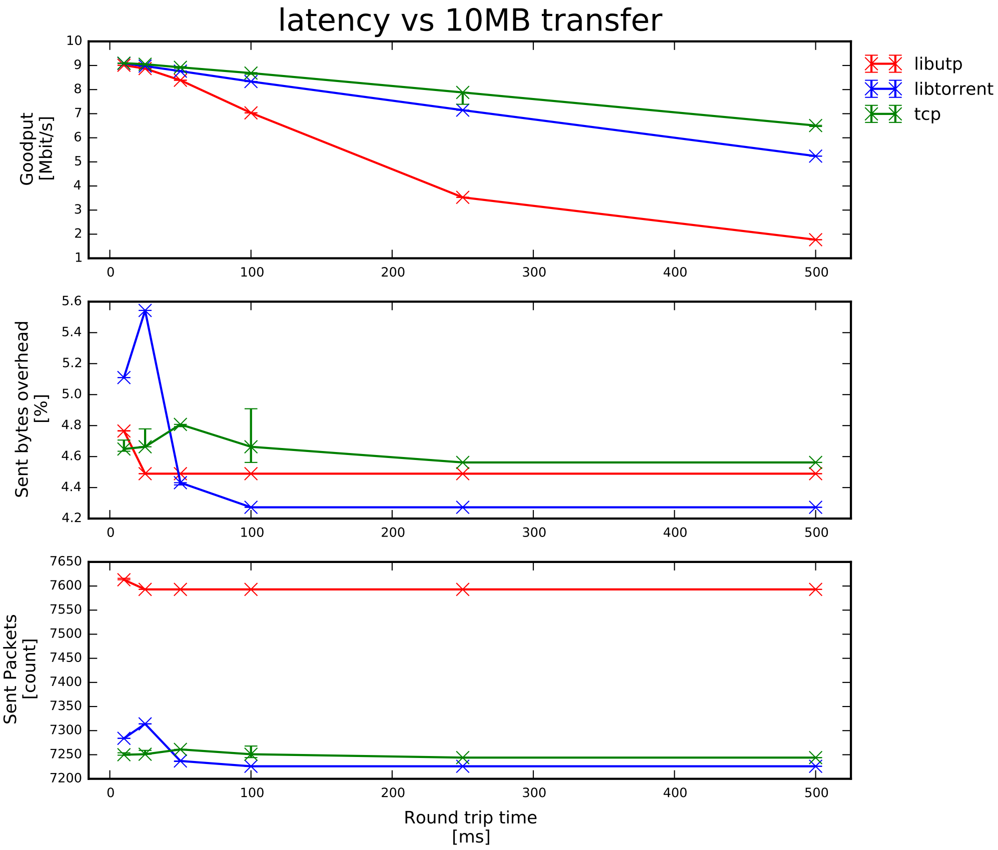


Figure 6.2.2: Latency test, 10 Mbit/s, 10-500 ms RTT, 0% loss

The packet loss test in figure 6.2.3 shows that libutp and libtorrent are not affected as must as TCP in case of packet loss.

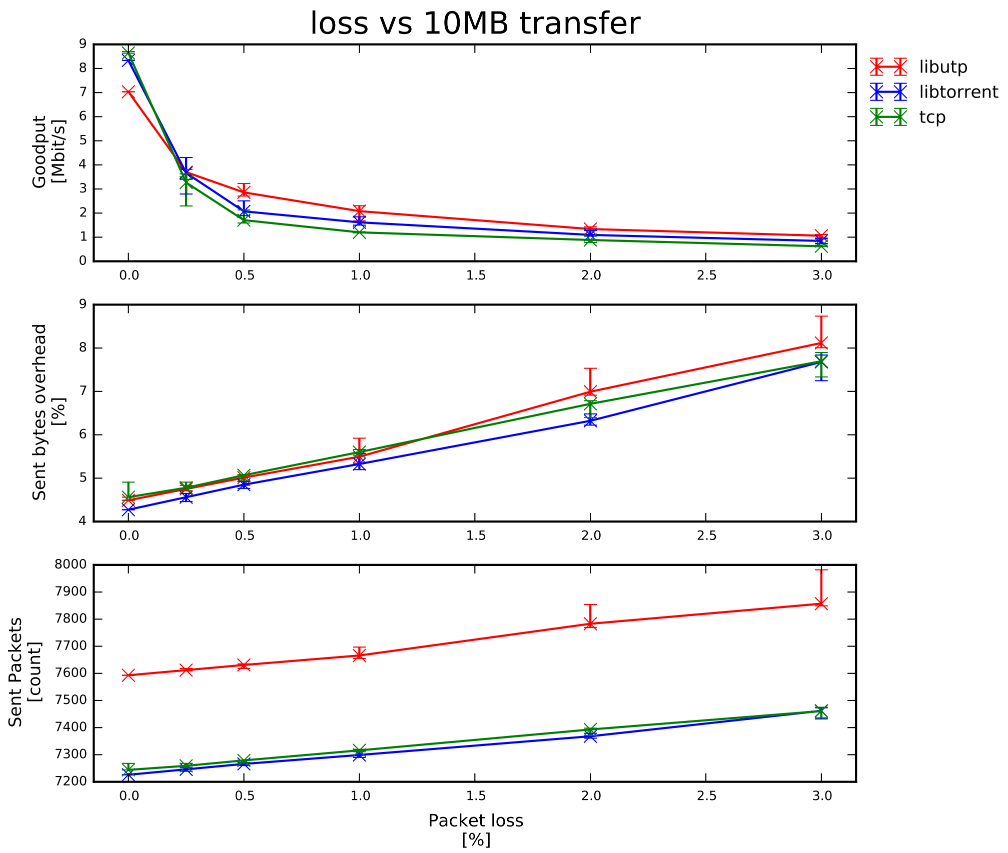


Figure 6.2.3: Loss test, 10 Mbit/s, 100 ms RTT, 0-5% loss

The delay jitter test in figure 6.2.4 shows how sensitive uTP is to jitter. It is not surprising since the one-way packet timings directly influence uTP's congestion controller and the transfer speed.

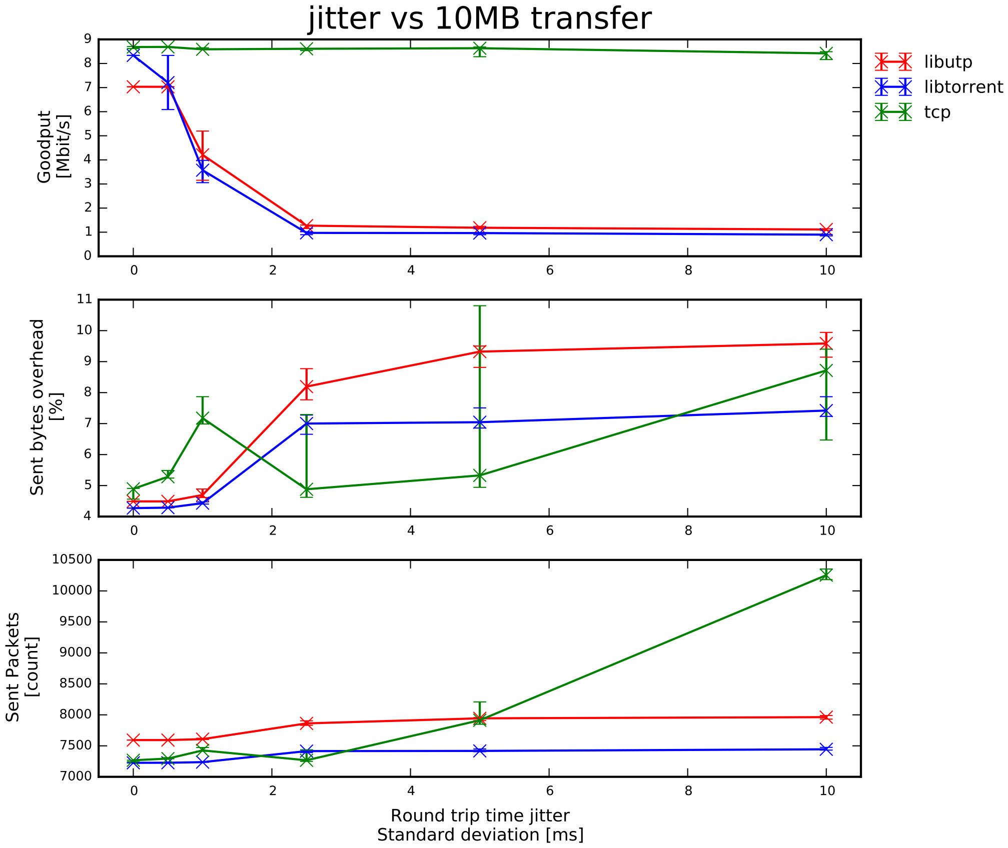


Figure 6.2.4: Delay jitter, 10 Mbit/s, 50 ms RTT, 0% loss, 0-10 ms std jitter

6.3 Congestion tests

The implementations were also tested to see how their congestion control reacts to concurrent TCP connections.

These tests transfer a large file with uTP and after 20 seconds, Iperf is executed for 10 seconds. After 40 seconds from the start, Iperf is again executed for 10 seconds. uTP should notice the increase in the queuing delay and slow down it's transfer rate.

The tests results also shows the current delay, based on extracting ping packets' RTT values using Tshark. Since RTT is 100 ms and TARGET queuing delay is 100 ms, the use of uTP should only increase the ping time to an average of 200 ms. While TCP will try to maximize the delay.

6.3.1 Congestion tests results

The test results show how both implementations manages to keep the queuing delay near or below 200 ms when TCP is not used. Both quickly reacts to concurrent TCP connections and drop their transfer rate.

One visible difference is the beginning, where libtorrent in 6.3.1 quickly grows to almost full speed and libutp in 6.3.2 takes significantly longer time to reach full speed.

Looking at the source code, libutp uses a slow-start state, but the code actually have a linear growth limit. It will only grow window-size up to a maximum of MTU bytes[1] when the full CWND size is acknowledged. This comment suggests another intention:

```
// true if we're in slow-start (exponential growth) phase
bool slow_start;
```

libtorrent uses slow-start in the exponential sense, where CWND will grow with the number of acknowledged bytes up to the current CWND size.

Between and after the two TCP transfers, the LEDBAT congestion control can be seen to linearly grow CWND. Because libutp uses a higher GAIN than libtorrent it rises faster in this case.

---

[1] get_packet_size()

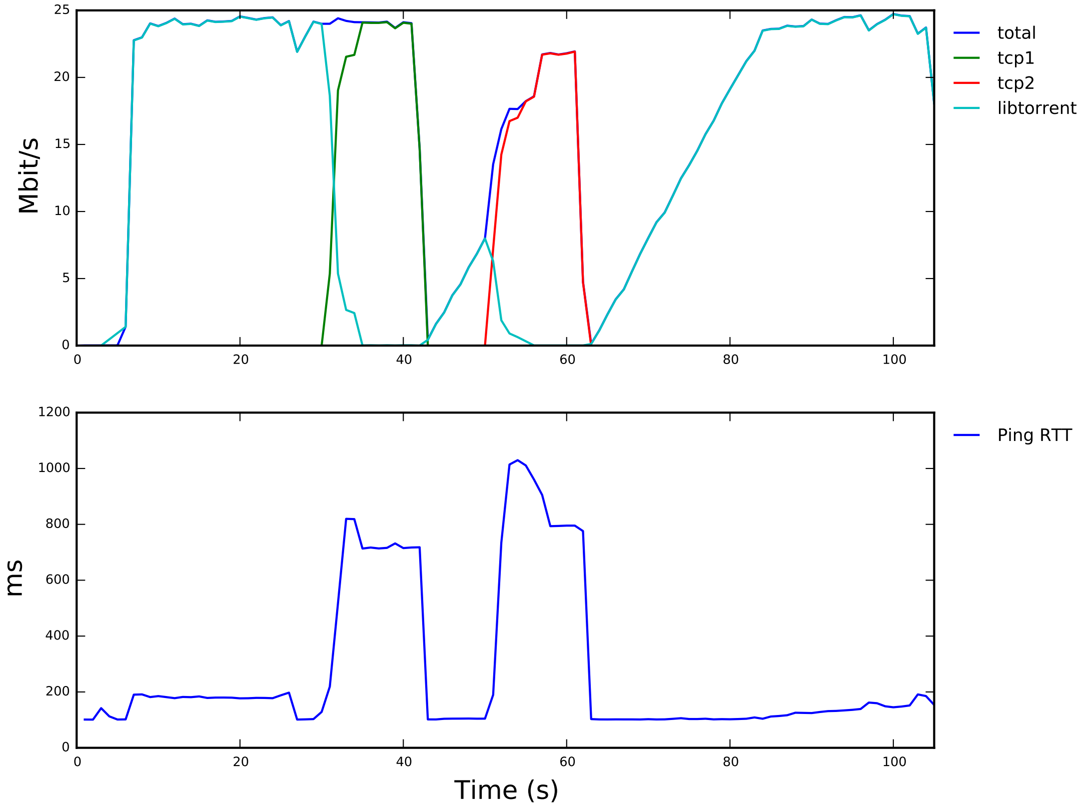


*Figure 6.3.1: Congestion test for libtorrent, bw=25 Mbit/s, RTT=100 ms, q=BDP*10*

6.4 Growing queuing delay

To measure the growing queuing delay issue, each implementation was tested with a long running transfer (60 minutes). To avoid overly large input and output files, the bandwidth limit was set to 1 Mbit/s.

The first test did not use any packet loss or other concurrent traffic sources.

The next test added a small amount of packet loss, 0.1%, to the network link to see if the issue persists.

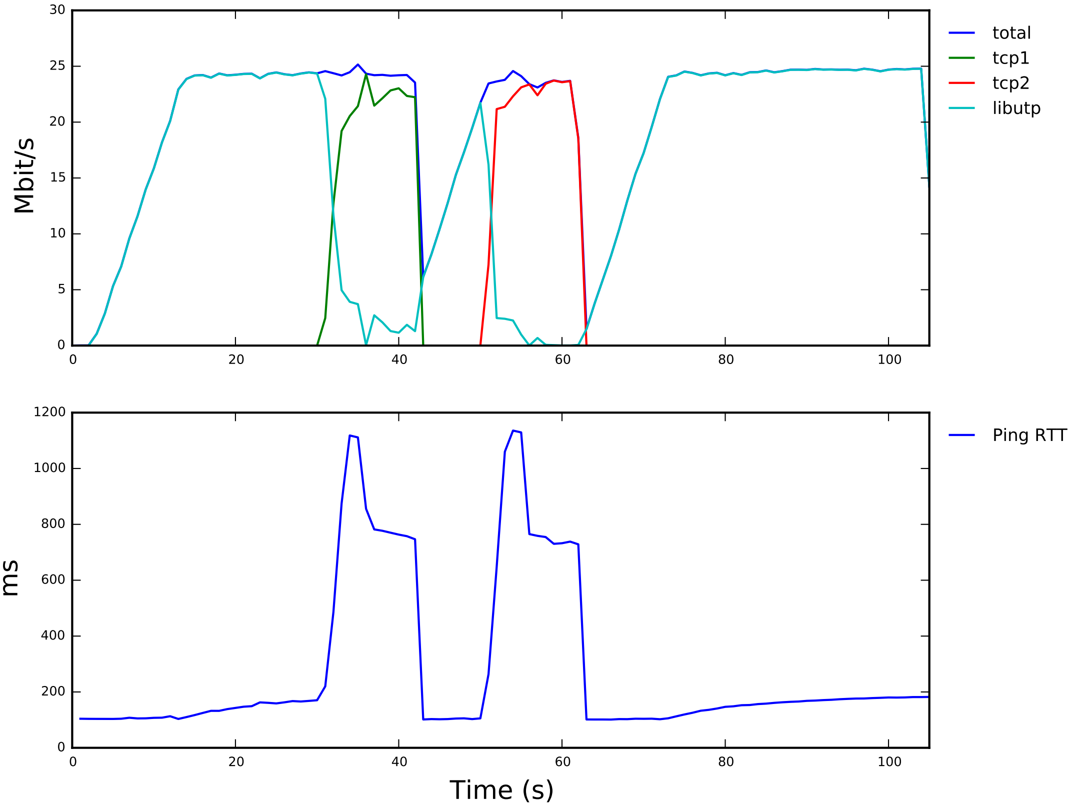


*Figure 6.3.2: Congestion test for libutp, bw=25 Mbit/s, RTT=100 ms, q=BDP*10*

6.4.1 Growing queuing delay results

The first test result for libutp is shown in figure 6.4.1, and libtorrent in figure 6.4.2. Both start with the expected delays (100 ms + 100 ms TARGET) but then they grow over time. libutp grows 100 ms every 13 minutes and libtorrent grows 100 ms every 20 minutes. This corresponds well to the code where libutp uses 13 entries in it's delay history[2], and libtorrent have 20 entries[3].

Adding packet loss, the both results are different and delay will stay close to 200 ms over the entire 60 minutes. See figures 6.4.3 and 6.4.4. Since the detection of packet loss leads to a great reduction in CWND, this causes a temporary stop in the transfer, which seems to be enough to measure the real one-way delay.

---

[2] utp_internal.cpp, delay_base_hist[DELAY_BASE_HISTORY=13]

[3] timestamp_history.hpp, m_history[history_size=20]

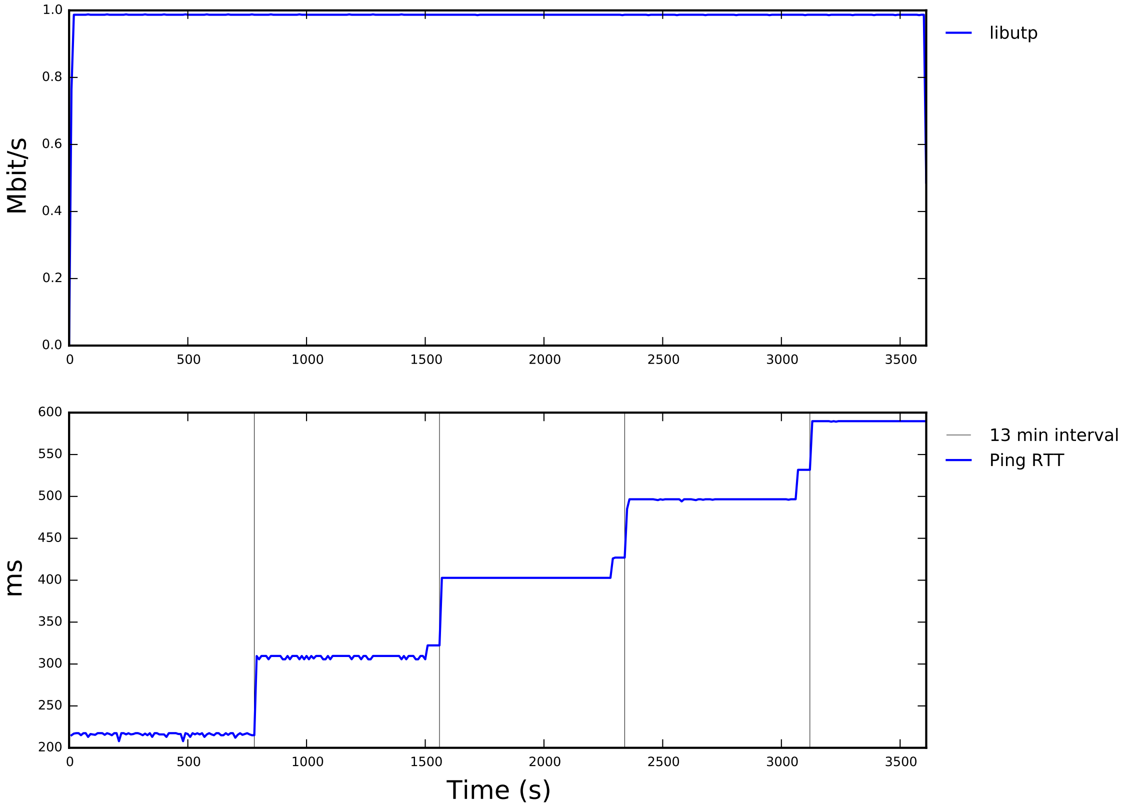


*Figure 6.4.1: Growing queuing delay test for libutp, bw=1 Mbit/s, q=BDP*10, RTT=100 ms*

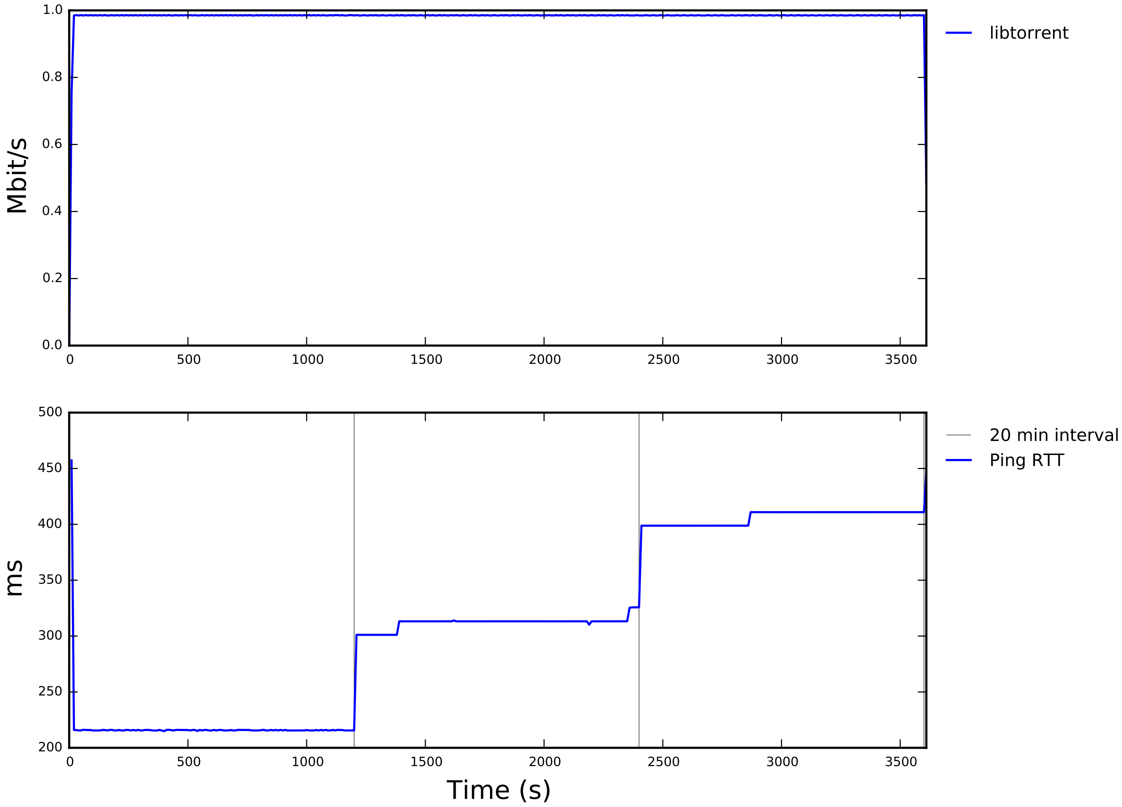


*Figure 6.4.2: Growing queuing delay test for libtorrent, bw=1 Mbit/s, q=BDP*10, RTT=100 ms*

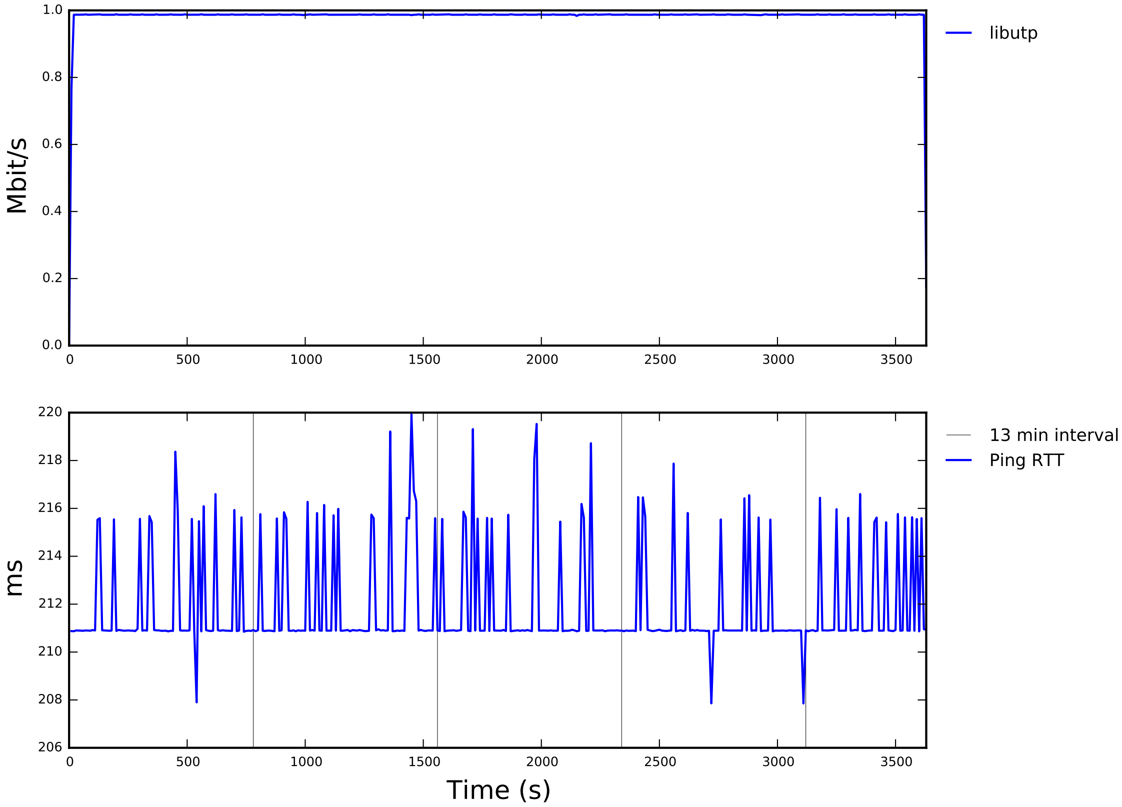


*Figure 6.4.3: Growing queuing delay test for libutp, bw=1 Mbit/s, q=BDP*10, RTT=100 ms, loss=0.1%*

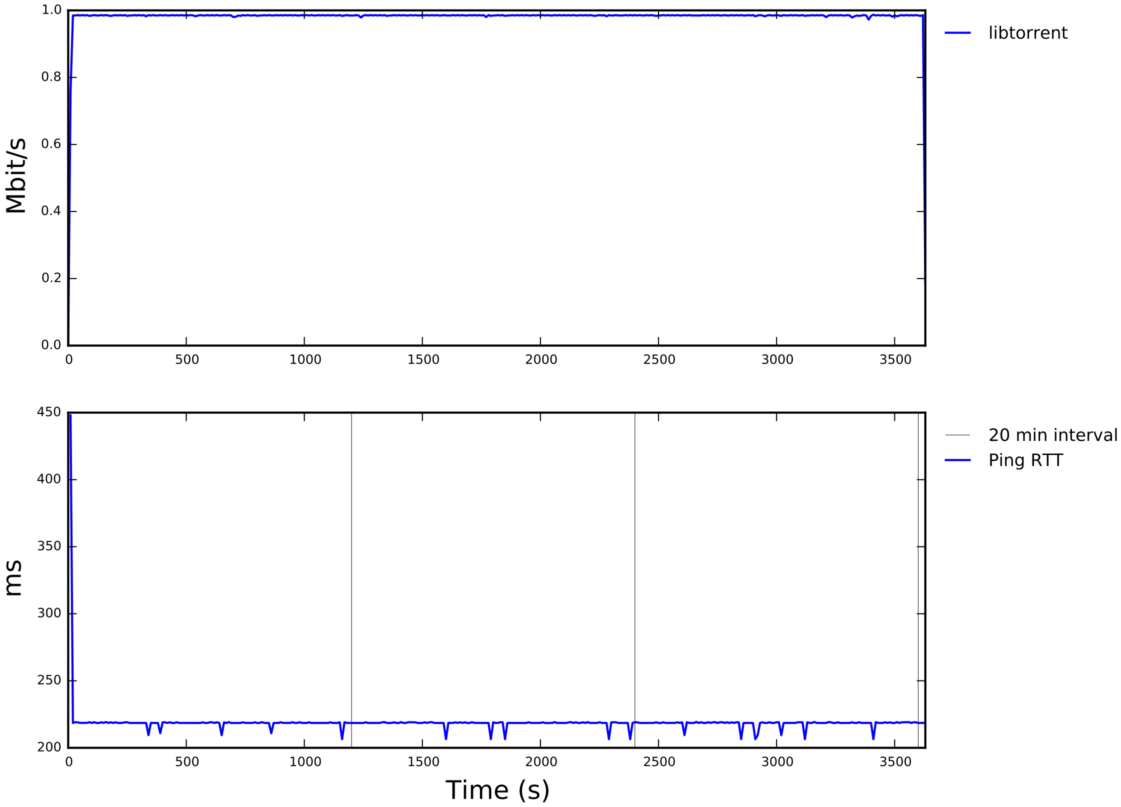


*Figure 6.4.4: Growing queuing delay test for libtorrent, bw=1 Mbit/s, q=BDP*10, RTT=100 ms, loss=0.1%*

6.5 Conclusions for uTP

The tests have shown the basic performance and congestion characteristics of uTP implementations. Similarities between the implementations can be seen, and not surprisingly also a few differences.

The performance of libtorrent was generally better than libutp, especially the high bandwidth tests, which is largely related to differences in their slow-start. This was clearly visible in the congestion plots and code, but if this is the intended behavior is unclear. Considering for TCP, that slow-start growth should be limited by $MIN(newly\_acked\_bytes, SMSS)$[49], it might be the other way around, and that libtorrent slow-start growth is too aggressive. This is listed as further work to clarify and resolve.

Compared to TCP, especially the libtorrent implementation can achieve almost matching performance across different network scenarios. The exception is delay jitter, which affects uTP performance greatly, since packet timings are used in its congestion algorithm. Another difference to TCP visible in the tests, is the reaction to concurrent connections where TCP tries to share the bandwidth fairly while uTP throttles back to accomplish its main goal of 'background transfer'.

In case of packet loss, libutp and libtorrent are even a little faster than TCP. This can be explained by differences in CWND reduction when loss is detected, where libutp has a limitation of reducing CWND once every 100 ms, while libtorrent's limitation is based on the current RTT estimation. TCP doesn't seem to have time-based limitations in its CWND reduction. To be faster than TCP when there is packet loss is not necessarily a good thing for congestion and fairness reasons.

Also shown is a reproduction of the growing queuing-delay issue in both implementations. This means the implementations don't have addressed this, for example by pausing transmission as suggested[42]. New experiments suggest that a small amount of packet loss will prevent the problem, which was not shown before. Since real networks are not completely lossless, this issue should not be a problem in practice.

Chapter 7

Conclusions

7.1 Discussion

The protocol study of Chapter 2 shows the basic requirements of reliable UDP protocols, the design goal and features of the studied UDP protocols. It is clear that they have much in common with TCP but each protocol also has some unique features. One common theme is that some of the lessons learned during the development of TCP are no longer optional, but now built into the UDP protocols, such as large windows and selective acknowledgments. Many features are closely aligned with the goal of the protocols, but native support for multiple streams to avoid head-of-line blocking are central for both SCTP and QUIC, which would be difficult to achieve with TCP.

A simple but perhaps the most important reason to use UDP-based transport protocols is that applications can select the protocol that best suits their needs. Also important to remember are some of the drawbacks, for example the implementations are a lot less tested and often use non-standard APIs.

Chapter 4 shows that while it's easy to get started with running network emulation in Mininet, an effort is still required to verify how things actually work. Both to check that Mininet is used correctly but also to verify and fix aspects of Mininet itself, such as packet loss and the issue with huge TCP packets. Even with initial verifications, the experience has been an iterative process where issues could still ”pop-up” at any stage.

For this thesis, two main test systems were built in order to perform the performance and congestion tests. Additional scripts that performed post processing and to plot graphs were also required. Now that the test systems exist, the actual tests can easily be repeated on updated versions of the implementations. Another possibility is using the test system to perform the same or similar tests on other UDP-based protocols, such as SCTP, which would only require suitable test-programs and minor changes.

One practical problem when performing emulated tests is to determine which scenarios to test and the exact parameters to use. It's easy to fall in the trap of wanting to explore the influence of a yet another parameter, leading to an endless number of tests to perform.

7.2 Conclusions

UDP-based protocols can avoid a number of limitations with TCP, such as: head-of-line blocking, the 4-tuple identification of connections and the dependency of the TCP version in the operating system.

New features of the studied UDP-based protocols have been described, such as QUIC's 0-RTT connection setup and how uTP's congestion control calculates one-way queuing delay. Other new features are briefly mentioned and not described in much detail in this thesis.

The work has shown how Mininet can be used to build systems to test UDP-based reliable implementations for multiple scenarios, from testing performance to congestion handling.

The main contributions of this thesis are the tests of QUIC and uTP implementations of their network performance and congestions handling.

For QUIC, the summary in Section 5.5 describes performance gains since a previous test by Connectify due to implementation improvements and verified good performance compared to TCP in case of packet loss due to FEC and when using multiple streams due to individual congestion control. Also found were scenarios that might require further analysis and improvements. The performance in case of delay jitter were much less than TCP, caused by its congestion control which seems unsuitable for wireless links. The low fairness in the case of two QUIC connection and two TCP connections suggests that QUIC obtains good performance at the expense of TCP.

The summary of uTP tests in Section 6.5 have shown similarities between the two implementations, such as the reaction to competing TCP connections, and differences, such as slow-start growth and time-based limitations of CWND reduction in case of packet loss. The uTP implementations came close to TCP's performance across the tested network scenarios except in case of delay jitter, caused by the essential usage of packet timings for uTP congestion algorithm. The previously identified problem with increasing queuing delay could be replicated in both implementations, but experiments indicate that a small amount of packet loss seems to prevent the problem.

When testing in isolated and emulated environments, the results should always be taken with a bit of skepticisms since real networks are influenced by many other factors and simultaneous traffic. To further verify the results of these tests, comparison with tests on real network are necessary.

7.3 Further work

Here are some limitations of this thesis that requires further work:

- From the results, there should be more analysis of what is caused by protocol properties in contrast to implementation details.
- Perform similar tests on real networks and compare the results.
- Using web browser based tests to measure how QUIC performs in terms of latency for page loads.
- The tested version of QUIC is already obsolete at this time. Automated tests using nightly-builds could be used to follow the development.
- Clarify if libutp uses too slow slow-start or if libtorrent is too aggressive, and if necessary send a patch to the library maintainer.

References

[1] R. Stewart, *Stream Control Transmission Protocol*, RFC 4960 (Proposed Standard), Updated by RFCs 6096, 6335, 7053, Internet Engineering Task Force, Sep. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4960.txt.

[2] 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), Internet Engineering Task Force, May 2013. [Online]. Available: http://www.ietf.org/rfc/rfc6951.txt.

[3] J. Roskind. Google QUIC design specification, [Online]. Available: https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34.

[4] L. Strigeus, G. Hazel, S. Shalunov, *et al.* (Jun. 22, 2009). uTorrent transport protocol (UTP), [Online]. Available: http://bittorrent.org/beps/bep_0029.html.

[5] B. Lantz and B. Heller. Mininet - an instant virtual network on your laptop (or other pc), [Online]. Available: http://www.mininet.org.

[6] Chromium. (Apr. 17, 2015). A QUIC update on Google's experimental transport, [Online]. Available: http://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html.

[7] G. Stein. (Jul. 29, 2013). This is why your bittorrent downloads move so fast, interview with Stanislav Shalunov, [Online]. Available: http://www.fastcolabs.com/3014951/why-your-bittorrent-downloads-move-so-fast.

[8] BitTorrent. Libutp - uTorrent transport protocol library, [Online]. Available: https://github.com/bittorrent/libutp.

[9] A. Norberg. Libtorrent website, [Online]. Available: http://www.libtorrent.org.

[10] J. F. Kurose and K. W. Ross, *Computer networking: a top-down approach*, Sixth edition. Boston: Pearson, 2013, ISBN: 978-0132856201.

[11] K. R. Fall and W. R. Stevens, *TCP/IP Illustrated, volume 1: The protocols*, Second edition. Addison-Wesley, 2011, ISBN: 978-0321336316.

[12] J. Postel, *User Datagram Protocol*, RFC 768 (INTERNET STANDARD), Internet Engineering Task Force, Aug. 1980. [Online]. Available: http://www.ietf.org/rfc/rfc768.txt.

[13] R. Braden, *Requirements for Internet Hosts - Communication Layers*, RFC 1122 (INTERNET STANDARD), Updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633, 6864, Internet Engineering Task Force, Oct. 1989. [Online]. Available: http://www.ietf.org/rfc/rfc1122.txt.

[14] B. M. Leiner, V. G. Cerf, D. D. Clark, *et al.*, "A brief history of the Internet", *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 5, pp. 22–31, 2009. [Online]. Available: http://www.cs.ucsb.edu/~almeroth/classes/F10.176A/papers/internet-history-09.pdf.

[15] V. Cerf, Y. Dalal, and C. Sunshine, *Specification of Internet Transmission Control Program*, RFC 675, Internet Engineering Task Force, Dec. 1974. [Online]. Available: http://www.ietf.org/rfc/rfc675.txt.

[16] V. Jacobson, "Congestion avoidance and control", in *ACM SIGCOMM computer communication review*, ACM, vol. 18, 1988, pp. 314–329. [Online]. Available: http://www.eecs.berkeley.edu/~sylvia/papers/congavoid.pdf.

[17] V. Jacobson, R. Braden, and D. Borman, *TCP Extensions for High Performance*, RFC 1323 (Proposed Standard), Obsoleted by RFC 7323, Internet Engineering Task Force, May 1992. [Online]. Available: http://www.ietf.org/rfc/rfc1323.txt.

[18] M. Mathis, J. Mahdavi, S. Floyd, *et al.*, *TCP Selective Acknowledgment Options*, RFC 2018 (Proposed Standard), Internet Engineering Task Force, Oct. 1996. [Online]. Available: http://www.ietf.org/rfc/rfc2018.txt.

[19] M. Belshe, R. Peon, and M. Thomson, *Hypertext Transfer Protocol Version 2 (HTTP/2)*, RFC 7540 (Proposed Standard), Internet Engineering Task Force, May 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7540.txt.

[20] T. Dierks and E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.2*, RFC 5246 (Proposed Standard), Updated by RFCs 5746, 5878, 6176, 7465, 7507, Internet Engineering Task Force, Aug. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt.

[21] T. Ylonen and C. Lonvick, *The Secure Shell (SSH) Transport Layer Protocol*, RFC 4253 (Proposed Standard), Updated by RFC 6668, Internet Engineering Task Force, Jan. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4253.txt.

[22] F. Audet and C. Jennings, *Network Address Translation (NAT) Behavioral Requirements for Unicast UDP*, RFC 4787 (Best Current Practice), Updated by RFC 6888, Internet Engineering Task Force, Jan. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4787.txt.

[23] S. Guha, K. Biswas, B. Ford, *et al.*, *NAT Behavioral Requirements for TCP*, RFC 5382 (Best Current Practice), Internet Engineering Task Force, Oct. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5382.txt.

[24] R. Stewart, M. Ramalho, Q. Xie, *et al.*, *Stream Control Transmission Protocol (SCTP) Partial Reliability Extension*, RFC 3758 (Proposed Standard), Internet Engineering Task Force, May 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3758.txt.

[25] S. Shalunov, G. Hazel, J. Iyengar, *et al.* (Dec. 1, 2012). Low extra delay background transport (LEDBAT), [Online]. Available: http://tools.ietf.org/html/rfc6817.

[26] A. Ford, C. Raiciu, M. Handley, *et al.*, *TCP Extensions for Multipath Operation with Multiple Addresses*, RFC 6824 (Experimental), Internet Engineering Task Force, Jan. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc6824.txt.

[27] D. J. Bernstein. (1996). Syn cookies, [Online]. Available: http://cr.yp.to/syncookies.html.

[28] R. Jain, D.-M. Chiu, and W. Hawe, *A quantitative measure of fairness and discrimination for resource allocation in shared computer systems*, DEC-TR-301, 1984.

[29] S. Floyd, M. Handley, J. Padhye, *et al.*, *TCP Friendly Rate Control (TFRC): Protocol Specification*, RFC 5348 (Proposed Standard), Internet Engineering Task Force, Sep. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5348.txt.

[30] R. Stewart, Q. Xie, K. Morneault, *et al.*, *Stream Control Transmission Protocol*, RFC 2960 (Proposed Standard), Obsoleted by RFC 4960, updated by RFC 3309, Internet Engineering Task Force, Oct. 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2960.txt.

[31] P. Karn and W. Simpson, *Photuris: Session-Key Management Protocol*, RFC 2522 (Experimental), Internet Engineering Task Force, Mar. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2522.txt.

[32] R. Stewart, Q. Xie, M. Tuexen, *et al.*, *Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration*, RFC 5061 (Proposed Standard), Internet Engineering Task Force, Sep. 2007. [Online]. Available: http://www.ietf.org/rfc/rfc5061.txt.

[33] A. Langley and W.-T. Chang. QUIC crypto, [Online]. Available: https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g.

[34] Google Chromium. QUIC wire layout specification, [Online]. Available: https://docs.google.com/document/d/1WJvyZflAO2pq77yOLbp9NsGjC1CHetAXV8I0fQe-B_U.

[35] Y. Cheng, J. Chu, S. Radhakrishnan, *et al.*, *TCP Fast Open*, RFC 7413 (Experimental), Internet Engineering Task Force, Dec. 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7413.txt.

[36] A. Langley. (Jun. 18, 2010). Transport layer security (TLS) snap start, [Online]. Available: http://tools.ietf.org/html/draft-agl-tls-snapstart-00.

[37] Connectify. (Nov. 7, 2013). Taking google's QUIC for a test drive, Connectify, [Online]. Available: http://www.connectify.me/blog/taking-google-quic-for-a-test-drive/.

[38] S. Shalunov, G. Hazel, J. Iyengar, *et al.*, *Low Extra Delay Background Transport (LEDBAT)*, RFC 6817 (Experimental), Internet Engineering Task Force, Dec. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6817.txt.

[39] TorrentFreak. (Aug. 14, 2009). Torrentfreak bittorrent client usage sep-2009, [Online]. Available: http://torrentfreak.com/utorrent-still-on-top-bitcomets-market-share-plummets-090814/.

[40] G. Carofiglio, L. Muscariello, D. Rossi, *et al.*, "A hands-on assessment of transport protocols with lower than best effort priority", in *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, IEEE, 2010, pp. 8–15. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5735831.

[41] S. Q. V. Trang, N. Kuhn, E. Lochin, *et al.*, "On the existence of optimal LEDBAT parameters", in *Communications (ICC), 2014 IEEE International Conference on*, IEEE, 2014, pp. 1216–1221. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6883487.

[42] D. Ros and M. Welzl, "Assessing LEDBAT's delay impact", *IEEE Communications Letters*, vol. 17, no. 5, pp. 1044–1047, May 2013, ISSN: 1089-7798. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6496997.

[43] B. Heller, "Reproducible network research with high-fidelity emulation", PhD thesis, Stanford University, 2013. [Online]. Available: https://stacks.stanford.edu/file/druid:zk853sv3422/heller_thesis-augmented.pdf.

[44] A. Jurgelionis, J. Laulajainen, M. Hirvonen, *et al.*, “An empirical study of netem network emulation functionalities”, in *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, IEEE, 2011, pp. 1–6.

[45] Y. Gong, D. Rossi, C. Testa, *et al.*, “Fighting the bufferbloat: on the coexistence of AQM and low priority congestion control”, *Computer Networks*, vol. 65, pp. 255–267, Jun. 2014, ISSN: 13891286. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S1389128614000188`.

[46] B. Ford, “Structured streams: a new transport abstraction”, *ACM SIGCOMM Computer Communication Review*, vol. 37, pp. 361–372, 2007. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1282421`.

[47] Y. Gu. UDP-based data transfer protocol, [Online]. Available: `http://udt.sourceforge.net`.

[48] E. Protalinski, *Google Chrome now has over 1 billion users*, May 28, 2015. [Online]. Available: `https://venturebeat.com/2015/05/28/google-chrome-now-has-over-1-billion-users/`.

[49] M. Allman, V. Paxson, and E. Blanton, *TCP Congestion Control*, RFC 5681 (Draft Standard), Internet Engineering Task Force, Sep. 2009. [Online]. Available: `http://www.ietf.org/rfc/rfc5681.txt`.

Appendix A

Test system details

Host system

- CPU: Intel i7-3520M 2.90 GHz (4 cores)
- RAM: 16GB RAM
- HDD: 180GB Intel SSD

Software versions:

- Ubuntu 14.04.1 LTS, 64-bit
- Linux kernel 3.13.0-43-generic Ubuntu SMP, CFS scheduler default, IPv6 disabled.
- Mininet 2.1.0 (Ubuntu packaged) with loss-fractional.patch.
- Matplotlib 1.4.2
- Nginx - 1.4.6
- curl - 7.35.0
- iperf - 2.0.5
- LaTeX- TeX Live 1.40.14
- RFC bibtex - 2015-06-17 from http://tm.uka.de/~bless/bibrfcindex.html

Tested UDP-protocol versions:

- QUIC - Chromium git checkout 2014-10-28, release compiled.
- libutp - git checkout 2014-05-20, release compiled.
- libtorrent - version 1.0.2, release compiled.

Appendix B

List of abbreviations

- ACK - Acknowledgement of arrived packets.
- ARP - Address Resolution Protocol, for resolving IP addresses on Ethernet.
- AIMD - Additive Increase Multiplicative Decrease
- ARQ - Automatic Repeat reQuest. Resends lost packets, often when used in wireless links.
- BDP - Bandwidth Delay Product. Used to calculate optimal window or buffer sizes, queue lengths or transmission delay.
- CWND - Congestion Window - The allowed number of unacknowledged bytes or packets. Controls the transfer speed.
- DoS - Denial of Service.
- DUP - Duplicate.
- DUP ACK - Duplicate acknowledgment.
- FEC - Forward Error Correction. Redundancy in transmissions to correct for certain errors.
- HTTP - Hypertext Transfer Protocol, used for web requests and responses.
- HTTPS - HTTP used over TLS.
- HOL - Head of line blocking - Problem when multiplexing streams over TCP.

- IP - Internet Protocol v4 and v6 - Network layer.
- OS - Operating System
- MSS - Maximum Segment Size. The maximum data size TCP will use. Often 1460.
- MTU - Maximum Transferable Unit. IP packets when sent over Ethernet often have the maximum size of 1500 bytes.
- NETEM - NETwork EMulator, component in Linux Traffic Control (tc). Used to emulate packet delay, delay jitter, packet loss and queue length.
- PCAP - Packet capture file. Typically recorded using tcpdump and analyzed with Wireshark or tshark.
- RTT - Round trip time - The time it takes for a packet back and forth been two hosts.
- SACK - Selective ACKnowledgement - Acknowledgement of multiple ranges of non-contiguous data.
- SCTP - Stream Control Transfer Protocol.
- SYN - SYNchronize flag used in TCP when establishing new connections.
- TC - Linux Traffic Control - subsystem of the kernel to
- TCP - Transmission Control Protocol. Connection oriented, reliable transport protocol.
- TLS - Transport Layer Security. Encrypted transport protocol used over TCP.
- uTP - uTorrent Torrent Protocol or Micro Torrent Protocol. UDP-based, reliable transport protocol for BitTorrent file sharing.
- UDP - User Datagram Protocol - Connection-less, unreliable transport protocol.
- VM - Virtual Machine.