DEGREE PROJECT

Network Test Capability of ModernWeb Browsers

Oskar Klang

Computer Science and Engineering, master's level
2019

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

Network Test Capability of Modern Web Browsers

Oskar Klang

Luleå University of Technology
Dept. of Computer Science, Electrical and Space Engineering

April, 2019

Abstract

Web browsers are being used for network diagnostics. Users commonly verify their Internet speed by using a website, Bredbandskollen.se or speedtest.net for example. These test often need third party software, Flash or Java applets. This thesis aims at prototyping an application that pushes the boundaries of what the modern web browser is capable of producing regarding network measurements, without any third party software.

The contributions of this thesis are a set of suggested tests that the modern browser should be capable of performing without third party software. These tests can potentially replace some of network technicians dedicated test equipment with web browser capable deceives such as mobile phones or laptops. There exist both TCP and UDP tests that can be combined for verifying some Quality of Service (QoS) metrics. The TCP tests can saturate a gigabit connection and is partially compliant with RFC 6349, which means the traditional Internet speed test sites can obtain more metrics from a gigabit throughput test then they do today.

Keywords: WebRTC, HTML5, network diagnostics, network test, speed test, RFC 6349

Preface

This thesis work was performed as the last task of the Master Programme in Computer Science and Engineering at Luleå University of Technology, LTU. The thesis work was conducted in the first half of 2015 and carried out at the company Netrounds in Luleå, Sweden.

I would like to express my gratitude to the people at Netrounds for this opportunity. Special thanks to Fredrik Kers and Marcus Friman who, amongst other guidance, gave valuable pointers to QoS marker testing, RFC 6349 and RFC 3357. I am also grateful for the valuable discussions about TCP with Peter Boström and C++ with Erik Alapää.

Oskar Klang

Contents

Chapter 1

Introduction

What do you do when you quickly want to verify the Internet connection speed your provider promise you? You have most likely used a website application that measures your speed. Speedtest.net[1] or bredbandskollen.se[2] for example. These websites usually measures three metrics, namely your download speed, upload speed and delay, with a simple click by the user. If you use a Mac, for example, the websites might not work since they rely on third party software that some platforms do not support, e.g. Linux, OS X, IOS, Andorid.

Today's network technicians occasionally also use these websites services but they often need expensive and sometimes dedicated hardware for measuring more metrics in order to locate network problems. If these websites can include more metrics could it save time for the technicians who may not even have to physically go out to the field to deploy hardware for the desired measurements.

With the rise of the new technologies in the modern web browsers these tests should be able to expand their measured metrics, without the use of third party software.

1.1 Goal

The goals of this thesis are to

- measure the throughput of a fully saturated gigabit connection with a modern web browser without third party installation, e.g. Flash or Java;
- construct theoretical tests that contain network metrics a modern web browser should be able to produce without third party installation;
- create a prototype that enables users to run these tests and measure various network metrics via a modern web browser; and
- evaluate how these tests perform on real hardware.

1.2 Scope

Given this thesis's time constraint it is important to limit the workload to the most essential components. The work will neglect the prototype's freeness from bugs and security as long the measurements are not affected. The user experience is not considered when developing the prototype.

There are countless numbers of possible hardware, operating systems and platforms that this thesis work could consider. But only the provided Windows 7 workstation will be considered when evaluating the tests and the prototype.

The prototype's server does not have a domain name, only IP address. This constraint on the prototype will get referred to when the browsers limit of TCP connections is discussed.

1.3 Related work

There exist a wide variety of solutions for measuring network quality, speed and giving diagnostic information. Those that are deployed via the browser are either too simplistic or uses additional software like Java or Flash. These solutions does not satisfy some users that either will not or cannot install or cannot run these additional software. Corporations for example may have strict systems for security reasons, some mobile devices does not support the software or they may be blocked by firewalls[3]. There were sadly no solution found that used the WebRTC technology for measuring the network quality.

Examples includes Bredbandskollen[2], NDT[4] and SG TCP/IP Analyzer[5]. Where Bredbandskollen test the upload speed, download speed and response time and is limited to Flash. These tree metrics are the most basic in the sense that they always are presented by similar solutions. And Bredbandskollen also seem to not be accurate because the download speed is always gives higher speed then is theoretical possible for my connection. This makes you wonder what quality this tool offer when this low accuracy is so obvious. Bredbandskollen can arguably be classified to being basic in both what metrics it offer and in what quality those metrics are.

NDT have the same tree metrics and was limited to Flash but have during this thesis work released a beta option for modern browsers. NDT additionally offer a advanced option where around one hundred TCP metrics are dumped at the user. These metrics are mostly made up of TCP statistics from the server kernel. Some of these metrics ended up to be used in this thesis but only because RFC6349 used them to give good insight into the network quality. Simply dumping a dozen of metrics alongside almost a hundred of others may be good for APIs but arguably not fur humans. The part from NDT that seem to be for humans are jitter, TCP receive window and packet loss that all are presented in a readable manner. These metrics was also used in this thesis and are arguably almost expected to exist in a TCP speed measurement tool. Another concern is that the metrics are not split up between the upload and download test. One must

read the NDT source code for understanding what test a metric belongs to. This made it frustrating to interpret the metrics and made it more important for this thesis to clearly splitting the entire download and upload test in this thesis.

SG TCP/IP Analyzer is different from NDT and Bredbandskollen in the sense that it gives information about the network quality limited to a test that only runs in less than a second. The test aims at helping users with TCP tuning and normal misconfiguration like MTU and TTL. This tool does not require any additional software like Flash. This analyzer did not influence this thesis in any way because its purpose is so specific. But is still yet another example of what the browser is capable of. This thesis aimed more at the throughput scenario.

CHAPTER 2

Theory

This chapter presents the theory used in this thesis.

2.1 Network Concepts

There exist many different networking protocols responsible for transporting the Internet's traffic from one computer to the next. These protocols have different characteristics and uses.

2.1.1 Path MTU

All computers network interfaces have its own configured maximum transmission unit, MTU. This includes the routers between the sender and receiver. A router can have different MTU configurations on its interfaces. The packets are split up into smaller packets when they are meant to travel through such a router where the outgoing interface have a lower MTU then the individual packets size. This packet splitting is called fragmentation. The reassembly of a fragmented packet usually occurs on the computer that is the packets final destination [6].

The MTU for the path between the sender and receiver is called path MTU.

2.1.2 TCP

The sender will know if the data was received by the receiver when the data was sent with the Transmission Control Protocol, TCP. The data is divided into packets when the data leaves the sending computer and goes into the network. TCP is a stream oriented protocol for this reason, meaning the user sends a stream and not individual packets. TCP reliability is maintained by the sender's and receiver's buffer that are negotiated when packets should be transmitted and retransmitted due to packet loss [7].

Modern TCP implementations uses Path MTU discovery that tries to find the path MTU in order to pick a packet size that avoids fragmentation in the network [8].

It will take time for packets to travel in the network when the sender and receiver are over great distance. This delay will impact the rate packets can be transmitted since the buffer needs to hold packets for retransmitting them if necessary. The impact delay has on the packet send rate can be countered by increasing the buffer sizes. Another way around delays impact on the send rate can be to increase the number of TCP sessions since each have their own receive and send buffers [9].

Flow and congestion control is used by TCP for regulating the rate packets are sent. These mechanisms are basically increasing the rate until the receiver for some reason report lost data, at which point the rate is lowered. There also is a slow start phase when the TCP session first start or when much loss is reported, this phase start at zero and slowly increase the rate. Many different and smart congestion control algorithms exist but they rely on the same principle. The goal of congestion control is to offer fairness between different TCP sessions, while flow control's goal is to maintain maximum send rate. It is still bound by the sender and receiver buffers that automatically can be changed by another algorithm called Auto-Tuning, which can increase the buffers during a TCP session [9].

2.1.3 UDP/SCTP

Delivery of packets is not guaranteed by the User Datagram Protocol, UDP. It is a simple protocol where there is no feedback to the sender. There is also no flow or congestion control, witch gives UDP the ability to flood the network [10]. UDP is a message oriented protocol contrary to TCP. The Stream Control Transmission Protocol, SCTP, is also a message oriented protocol but the behavior of this protocol can be configured to resemble both UDP or TCP [11].

2.1.4 RTP

Real-time transport protocol, RTP, is often used for delivering video and audio. It is usually accompanied by its sister protocol real-time transmit control protocol, RTCP, for adjusting the streams quality. RTCP contain much useful information about the quality of the RTP session, such as jitter, delay, and loss [12].

2.1.5 Quality of Service

The quality of service, QoS, refers to the overall performance of the network. If the network would like to prioritize traffic that usually is indistinguishable from other traffic the packets can be modified to include a Differentiated Services Code Point, DSCP, value that indicate the priority of the session. This value can sometimes be changed by the network based on what port the connection is established over [13]. A example of such

traffic could be if a user downloads a file over the same network that the same user is engaging in a conference call with. The DSCP value can then indicate to the network that the conference call should get prioritized over the file download [14].

2.1.6 TCP Throughput Measurements

Data will be put into the payload of a wrapping packet that the network uses for delivering the packet to the receiver. The entire packet size can be used in the calculations or only the payload when measuring how much data is put through the network, hence throughput. Throughput will be calculated only with the payload in this thesis, also called goodput [9].

There are many different ways of measuring throughput. The theoretical formula for maximum achievable throughput

$$T = \frac{RWND \cdot 8}{RTT} \tag{2.1}$$

can be used for approximating the throughput by using the TCP receive window, RWND, and round trip time, RTT. The TCP receive window is how many bytes the receiver and sender can have traveling in the network, it is reflected by the senders and receivers buffer sizes. The RTT is the time it takes for packets to travel from the sender to the receiver and back again [9].

The throughput can also be measured simply by counting the data the receiver receive during a timespan. The sender can also, maybe a little counter intuitive, measure the throughput if the sender's buffer is kept full. The sender can this way measure the same rate the network have, hence the actual throughput. How the rate is the same for the sender as the receiver is illustrated in figure 2.1.

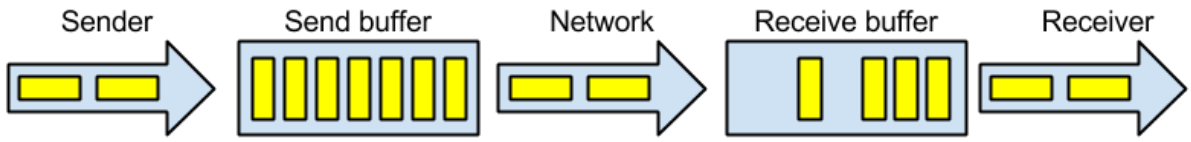


Figure 2.1: When the sender sends yellow packets to the receiver and keeps the send buffer full.

Some approximation must take place for aggregating the results when measuring multiple streams that report their throughput at different timestamps. The other streams will congest the network if there is loss for one stream, thanks to congestion control. More TCP sessions will therefore congest the network faster [9].

A distinction must be made before calculating with actual numbers. This distinction is between KB and kB that refers to 1024 bytes and 1000 bytes respectively.

There exist a default and maximum size for TCP's send and receive buffers that vary for different machines. Windows 7 for example have 64KB as default buffer size to start with and 16MB as maximum buffer size [15]. 64KB would deliver a maximum throughput

$$T = \frac{RWND \cdot 8}{RTT} = \frac{641024 \cdot 8}{10} = 52,4288 Mbit/s \tag{2.2}$$

when dealing with a round trip time of 10 ms. When flow and congestion control have fully utilized the buffers is the maximum achievable throughput

$$T = \frac{RWND \cdot 8}{RTT} = \frac{1610241024 \cdot 8}{10} = 13421.77 Mbit/s \tag{2.3}$$

if both the sender and receiver have buffers size of 16MB and have a round trip time of 10 ms.

2.2 Network metrics

The traditional tests, Bredbandskollen for example, usually measure only three metrics. Namely throughput in both directions and delay. Description of these and more metrics that will be used in this thesis is found in this section.

2.2.1 Round-Trip time

When measuring delay in a network is it referring to the time it takes for one client to send to the other. The round trip time, RTT, is referring to how long it takes for a message to travel between two clients back and forth [9].

2.2.2 Receive and Send Buffers

The receive and send buffers are interesting because they can act as a bottleneck for the throughput [9]. This bottleneck can be eliminated or found by the user if these metrics is presented.

2.2.3 Packet Loss

Packets can be considered lost for many different reasons. For example it may arrive to the intended receiver but the packet arrives too late. RFC 3357 presents the loss distance and loss period metrics that can be derived from simply recording if each sequential packet was lost or not. The loss distance metric describes how many packets that arrived without interruption by loss. While loss period describes how many packets that was lost in a row for a specific interval [16].

2.2.4 Transfer Time Ratio

The Transfer Time Ratio metric is defined in RFC 6349 as

$$TRT = \frac{ActualTCPTransferTime}{IdealTCPTransferTime} \tag{2.4}$$

where the first parameter is self explanatory and the second is derived from the physical networks capabilities. The Ideal Transfer Time

$$ITT = \frac{Transferredbits}{TCPPacketSize \cdot \frac{LinkSpeed}{PhysicalPacketSize}} \tag{2.5}$$

where the difference between TCP packet size and Physical packet size is the additional information the network uses to deliver the TCP packet to the intended receiver. The Link speed is basically the maximum speed capabilities of the network, in the unit of bits per second [9].

2.2.5 TCP Efficiency

In RFC 6349 is the TCP Efficiency metric defined as

$$E = 100 \cdot \frac{TransmittedBytes - RetransmittedBytes}{TransmittedBytes} \tag{2.6}$$

where the parameters are self explanatory. The metric represents how many percent of the transferred data had to be retransmitted due to loss [9].

2.2.6 Buffer Delay

In RFC 6349 the Buffer Delay metric is defined as

$$BD = 100 \cdot \frac{AverageRTTduringTransfer - BaselineRTT}{BaselineRTT} \tag{2.7}$$

where Baseline RTT is referring to the lowest measured RTT when the network is not congested [9].

2.2.7 Jitter

Jitter is an important aspect of the systems quality when dealing with time critical information over the network. Real time audio and video are such examples. It is defined in RTCP as

$$interarrivalJitter = abs((Rj - Ri) - (Sj - Si)) = abs((Rj - Sj) - (Ri - Si)) \tag{2.8}$$

where j and i are the latest two packets respectively, R is the time of arrival and S the time the packet was sent. This calculation should be made for each packet and then can averages be calculated for specific intervals, the entire connection for example [17].

Jitter over TCP does not say as much as it does over UDP since TCP have inherent jitter. Loss for example does introduce jitter in TCP since it then retransmits. Some kind of jitter can be represented by the RTT variation witch is the difference between maximum and minimum RTT during a TCP session. The RTT variation is estimated in the inner workings of all TCP implementations [18].

2.3 Browser Technologies

Web browsers communicates through the Hypertext Transfer Protocol, HTTP. This allows the browser to retrieve and send information to and from web servers. All web browsers today can also run code that is received from web servers, this code is in the JavaScript language. The browser can dynamically, without the need of additional software, open up further communication with web servers by using JavaScript.

2.3.1 AJAX

AJAX, asynchronous JavaScript and XML, allows the browser to make multiple HTTP requests using JavaScript. The JavaScript API for AJAX include continuous feedback at high frequency on how many bytes the request have loaded, which eliminates approximation when aggregating the streams throughput. All browsers have AJAX support [19].

The HTTP/1.1 standard states that a "single-user client SHOULD NOT maintain more than 2 connections with any server" [20]. This can be countered by using a wildcard domain for the server. The next constraint is the maximum number of total connections a browser will permit, there exist no workaround of these except maybe enabling some experimental flags in the browser. The Firefox browser have for example the Network.http.max-connections variable [21]. There exist a tool called Browserscope that gathers information where a summary can be seen of what different browsers capabilities are. Most major browsers support up to six simultaneous connections per host to [22].

2.3.2 Websockets

The browser will first make a HTTP request to the server that in turn upgrades the HTTP request to a websocket stream. The server and browser can negotiate extensions for websockets during this handshake, including compression for example [23]. It is important to disable this compression when measuring the throughput since it then would not measure the networks capability.

The JavaScript API only inform the receiver when the entire message have arrived,

nothing about the status of messages that are loading. The sender does have a variable for how much data it is in the send buffer and the network [24].

All modern browsers have support for websockets, and most have a very high limit on the number of simultaneous connections. Firefox, for example, have 200 [25].

2.3.3 WebRTC

This API gives browsers the ability to communicate directly between each other. The communication channels are audio, video and data. Where audio and video operates on secure RTP (SRTP) and secure RTCP (SRTCP). SRTCP reports QoS statistics about the session that most notably contains lost packets, duplicate packets, minimum jitter, maximum jitter, mean jitter and standard deviation jitter [17]. But the only way of transmitting video over the SRTP from a browser are from the user's video camera. Audio can be generated from the browser without an external source with the Web Audio API. This is not yet possible for video at the moment [26].

The Data Channel operates on SCTP over UDP where it is possible to change the SCTP's setting for reliable and ordered delivery via the JavaScript API [27]. There is obligatory encryption between two data channels [28].

2.4 Linux TCP Socket

The linux TCP socket have a option for retrieving information about the TCP session. This is retrieved as a TCP_INFO struct. Many metrics are included in the TCP_INFO struct that the underlying layer continuously update for each TCP session. Many of these variables are not accumulating but simply reflect the current snapshot of the TCP session. Many of the descriptions of these variables were derived through reading the source code or by conducting tests on real hardware. See table 2.1.

TCP_info variable	Description
tcpi_unacked	Number of not ack:ed transmits. Number of packet "in flight".
tcpi_last_data_sent	Timestamp delta when last data was sent
tcpi_last_data_recv	Timestamp delta when last data was received
tcpi_last_ack_recv	Timestamp delta when last ack was received
tcpi_pmtu	Path maximum transmission unit
tcpi_rtt	Smoothed round trip time [29] $SRTT = RTT$ $SRTT' = (7/8) \cdot SRTT + 1/8 \cdot RTT'$
tcpi_rttvar	Round-trip time variation [29] $RTTVAR = RTT/2$ $RTTVAR' = (3/4) \cdot RTTV + 1/4 \cdot \lvert SRTT - RTT' \rvert$
tcpi_rcv_ssthresh	Receiving slow start size threshold
tcpi_snd_ssthresh	Senders slow start size threshold
tcpi_snd_cwnd	Senders congestion window size
tcpi_rcv_rtt	Receivers RTT estimation
tcpi_rcv_space	Receiver queue space
tcpi_total_retrans	Retransmission for the entire connection

Table 2.1: Some of the variables in the TCP_INFO struct.

CHAPTER 3

Methodology and Implementation

This chapter describes how this thesis work was performed. The first phase describes how network metrics were identified by evaluating the modern browser. Next is how the prototype proposal and implementation phases were conducted. Finally the prototype evaluation phase is described.

3.1 Browser evaluation

It was a big part of the thesis to investigate what network metrics the modern browser is capable of measuring between one browser and one server. The initial step was to investigate the different ways of generating data on the network from and to a browser. The alternatives considered was AJAX, WebSockets, WebRTC audio, WebRTC video and WebRTC datachannels. Where WebRTC video could be eliminated since the browser would require access to the camera. The audio part of WebRTC could however be generated on the client side without the use of any microphone.

When there were enough information about the browsers limits the repository of Internet standards, maintained by the Internet Engineering Task Force (IETF), were searched for inspiration on what sort of metrics should and could be obtained. My supervisors at Netrounds also made suggestions. The two standards RFC 6349 [9] and RFC 3357 [16] was given much attention and would come to heavily influence this thesis. RFC 6349 supply additional metrics and methods for TCP throughput testing that explains different bottlenecks and is extremely applicable for a browser based test for understanding where these bottleneck are.

The investigation of what the server were capable of was started last. The browsers limitations and what sort of metrics that were interesting turned the attention to the TCP_INFO stuct and a native WebRTC client. Where the TCP_INFO struct can give

great insight into many interesting metrics when fully saturating a link. But the WebRTC client can give similar metrics and also more time critical metrics with low rates from both the RTP and SCTP protocols. Both these methods were chosen as input to the next phase since both these methods complemented each other.

3.2 Prototype proposal

This phase ties together the results from the previous phase into concrete tests. These tests makes up a prioritized list of different items that should be implemented and evaluated. It was this way clear which tests should be implemented and evaluated if the thesis needed to be limited in scope. The list of tests also served as a clear definition on what to implement and the supervisors at Netrounds could influence the work early.

One of the goals of the thesis was always to saturate a gigabit connection. The focus was made on throughput and QoS while defining these tests. The WebRTC Audio transport method did not get included in any of the tests. The WebRTC datachannel method was chosen instead as it covers the jitter and delay measurements of a unreliable transport protocol. The WebRTC Audio could offer more precise measurements since it is done at the RTP layer. But that also means it would be highly unlikely to come close to a saturated gigabit connection with WebRTC Audio.

The following example shows how the defined tests are presented. Each test indicates the test's input, name, and output. Some tests are reused in others. A short description of how the test can be implemented in theory is also included, see figure 3.1 for an example.

What all these tests have in common are the host and port input.

Figure 3.1: The template for a test named Test, where the host and port is the input and the output is indicated.

3.2.1 Baseline Test

This test can be implemented by opening requests to the server in a serial manner for as many samples that is desired. The actual RTT can be measured within a TCP session. All the samples can be used for an average RTT, see figure 3.2. The purpose of this test is to measure the delay when the connection is not saturated, the metric is defined in

RFC 6349.

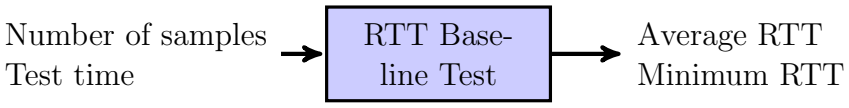


Figure 3.2: The RTT Baseline test.

3.2.2 Throughput Test

This test can be implemented with AJAX or Websockets. Each implementation should have the download and upload tests separated, see figure 3.3 and 3.4 respective. The minimum RTT is handled as an input to the test. The number of TCP sessions is also a input since different browsers have different limits. And the bandwidth input is used for calculating the TCP Eficcency that is defined in RFC 6349. The output is gathered from the TCP session(s) and presented to the end user via a separate communication channel. Note that there are different inputs and outputs depending on the direction of the test. This is because the browser is a limiting factor when sending traffic and gathering metrics. The server can for examle set DSCP values on the packets it sends when the browser cannot. The inputs and outputs are also different because the TCP_INFO struct gathers different information in different directions of the traffic.

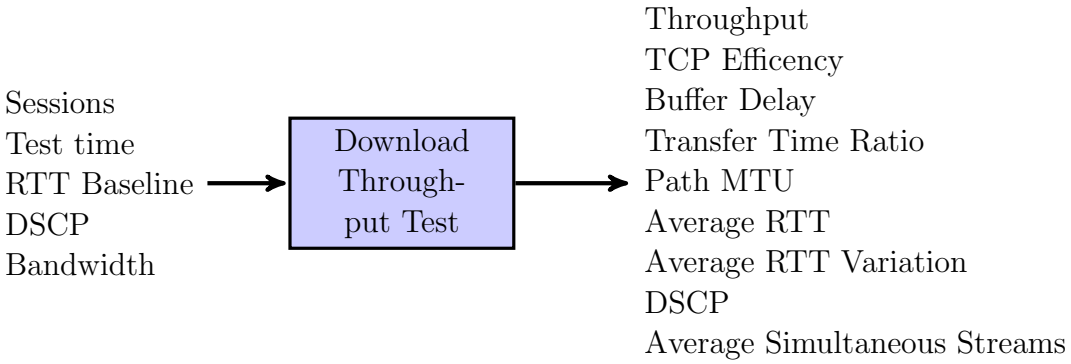


Figure 3.3: The Download Throughput test.

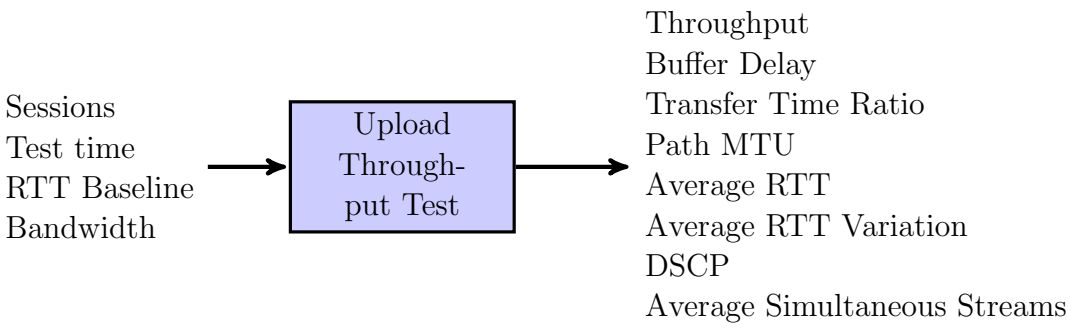


Figure 3.4: The Upload Throughput test.

3.2.3 Quality Test

This test can be implemented in the browser by configuring SCTP in WebRTC datachennel to resemble UDP very closely by configuring with unordered and unreliable delivery. The sequence number and timestamp must be in the payload so they can be compared these on the receiver for calculating the metrics. This assumes that the sender and receiver clocks are synchronized. Alternatively the RTT and jitter could also be calculated based on the RTT by reflecting the packets immediately when received by the receiver.

The metrics for lost packets and packets out of order can also be calculated on application layer. The upload and download test are separate, see figure 3.5 and 3.6 respective. This test should in theory be able to maintain high throughput since SCTP have flow control.

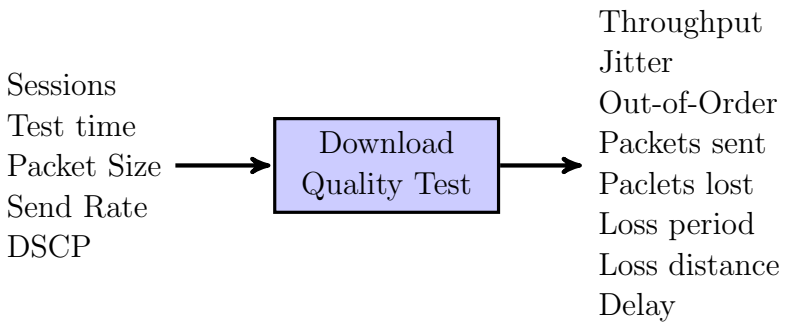


Figure 3.5: The UDP Upload Quality test.

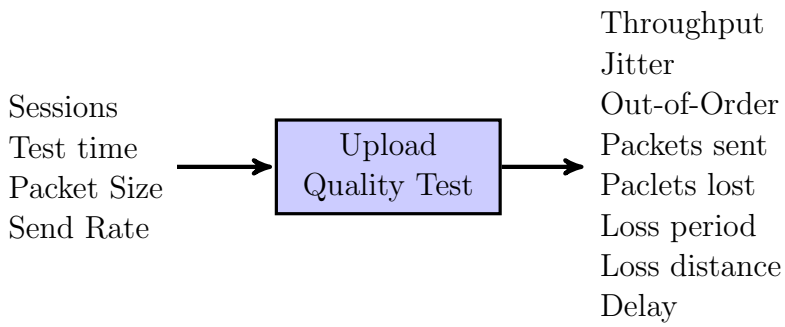


Figure 3.6: The UDP Download Quality test.

3.2.4 QoS Classes Test

The point of this test is to compare the output of multiple simultaneous instances of tests previously described. A user would for example be able to use this test to verify a home routers QoS settings for prioritizing gaming traffic over the download of a big file. This test can be implemented by aggregating the throughput and quality tests where it is meant the user changes the QoS markers for each test. The download test have the option of setting DSCP markers. Note the port input is important since this may be used by some as a QoS marker. The download and upload tests are separated, see figure 3.7 and 3.8 respective.

3.3 Prototype implementation

The ambition of this phase was to implement the tests that ware defined in the previous phase. The prototype would be a web server that not only pushes JavaScript code to the browser. The server would also have a back-end that can accept traffic generated by the code at the browser, generate traffic to the browser and measure metrics from the traffic.

The noteworthy problems that was encountered during implementation are presented and discussed in this section.

3.3.1 Development Environment

The workstation and the server was the same machine, a Linux distribution with 8 GB ram and 4 physical CPU cores. The outgoing traffic on the interface was manipulated by the netem emulator [30] so the prototype could be tested during development with different network scenarios. In the last days of implementation a Windows 7 machine were introduced that also could manipulate its outgoing traffic with a similar tool to netem called clumsy 0.2. The server was always deployed on the Linux machine. The

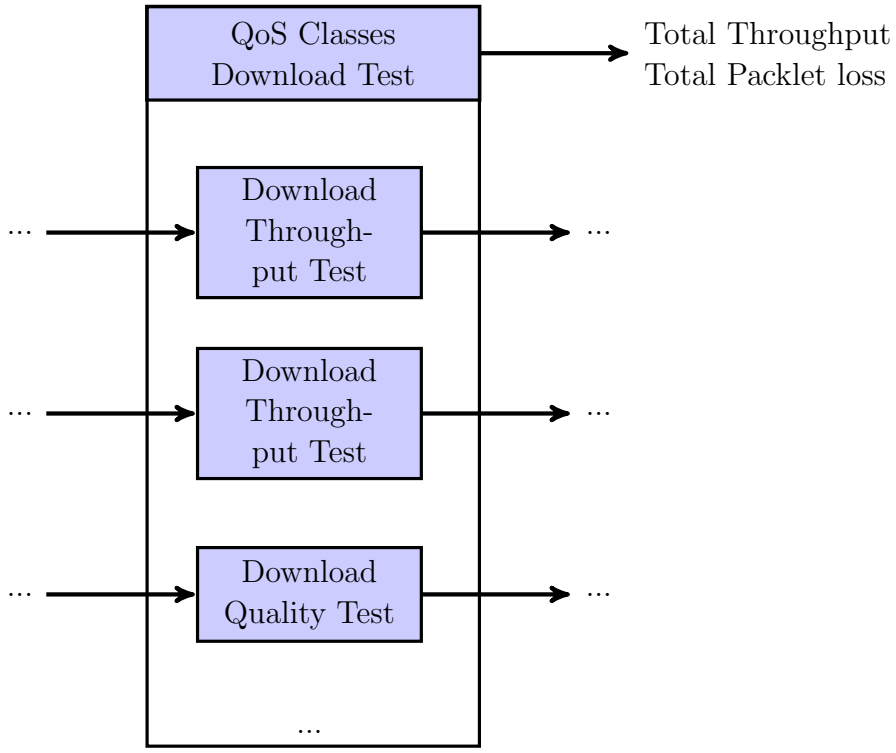


Figure 3.7: The QoS Classes Download test.

browsers Chrome and Firefox were also part of the development environment on both the Windows and Linux machines. Internet Explorer and Opera browsers were only installed on the Windows machine. The prototype were made to support all of these browsers on their respective platforms. There was always a possibility to add more platforms to the development environment, such as Mobile phones, more Windows versions or other platforms for the server. This was deemed out of scope of this thesis.

3.3.2 AJAX

There was a small differences, implementation wise, between AJAX and Websockets from the server's perspective. Additional parsing of the Websocket encapsulation and the Websocket handshake needed to be implemented. This small difference was one of the motivation for implementing both AJAX and Websockets for evaluating them alongside each other. It was also desirable to investigate the impact on the throughput measurement when the AJAX method have a limitation, posed by the browser, on maximum connections. This limitation could not be worked around by using a wildcard subdomain name, since it is a requirement for this prototype not to use a wildcard subdomain. This limitation may not be a problem if the TCP session could be open long enough

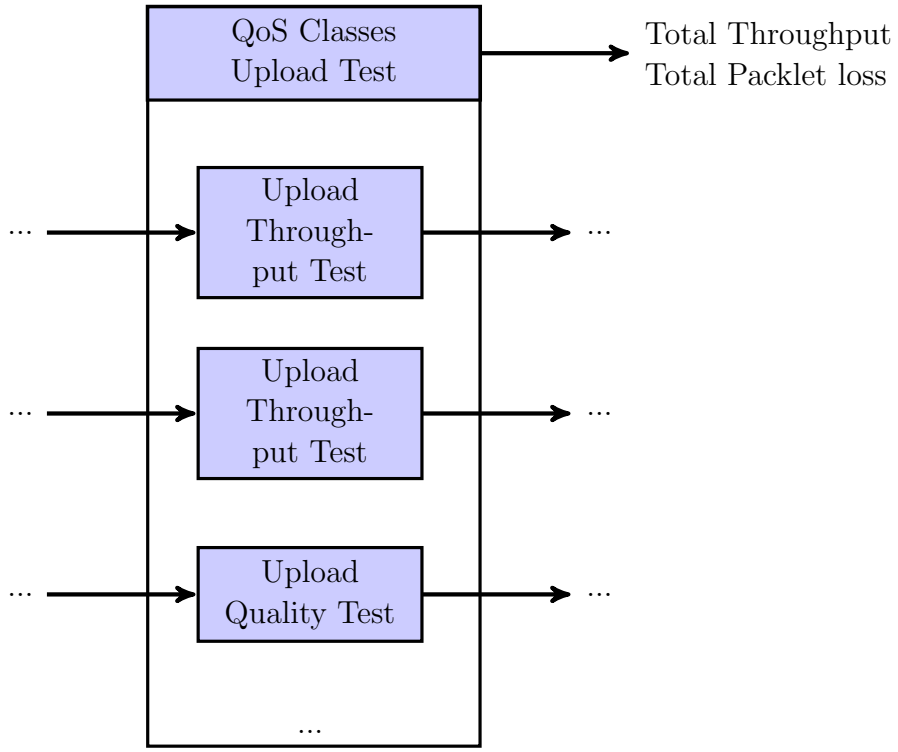
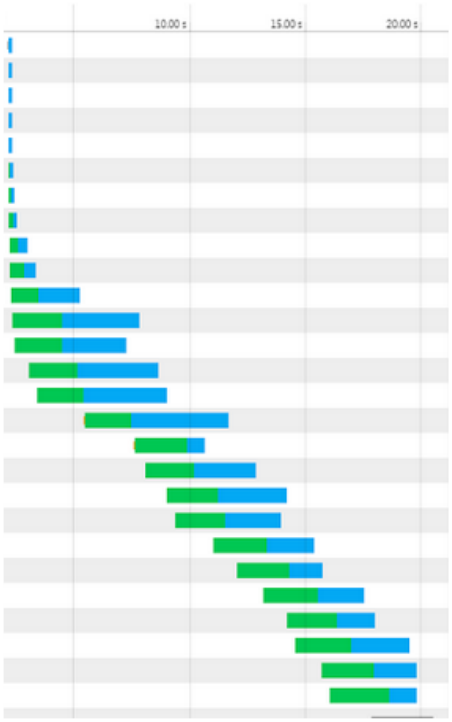


Figure 3.8: The QoS Classes Upload test.

for Auto-Tuning to increase the sender and receiver buffers. But some browsers may have limits on Auto-Tuning. The problems of the system's efficiency was apparent while implementing the throughput tests. Both the data generation on the client and server were time critical or could crash the browser. The effect of inefficiency can be seen in figure 3.9. The throughput was almost half of what it should be since so much of the TCP session's time was spent generating the payload.

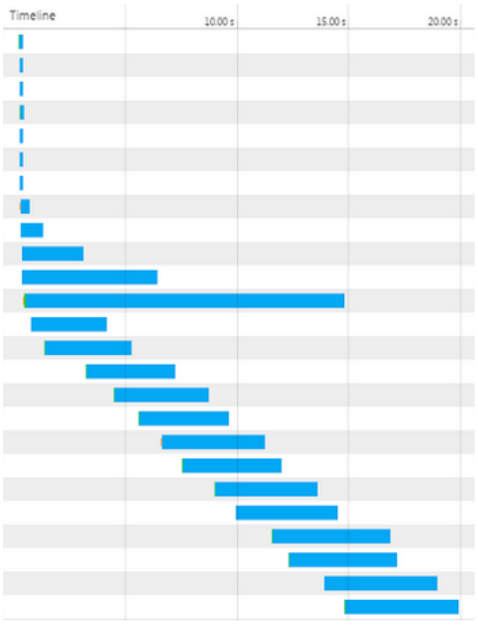
Figure 3.9 does not only give a feeling of the impact an inefficient implementation can introduce. The figure also shows how the results can be affected by high delay and relatively small payloads, note that it is not five parallel active streams in the left figure.

Google Chrome [31] Developer tools is a powerful tool that among other things can throttle the throughput and visualize the concurrent connections. This tool was used for keeping the throughput under 30 Mbit/s and used for verifying that the throughput calculations based on the JavaScript API reports is accurate. A test with a setting of a single parallel TCP stream was used. The result of the throughput calculations based on the continuous feedback from the JavaScript API can be seen in figure 3.10. The Throughput was stable between 30 and 35 Mbit/s.




```
std::string genData(int len) {
 data.reserve(len);
 for (int i=0; i<len; ++i)
  data += "a";
 return data;
}
```

(a) Slow download test




```
std::string genData(int len) {
 return std::string(len, 'a');
}
```

(b) Fast download test

Figure 3.9: Two different download tests of many TCP streams with a setting of 5 parallel session at any time. The payload increases by each session. The green color represents the time the server handles the request. The blue color is the download time. This is derived by using Google Chrome development tool. The C++ code shows the difference in the implementation for respective measurement.

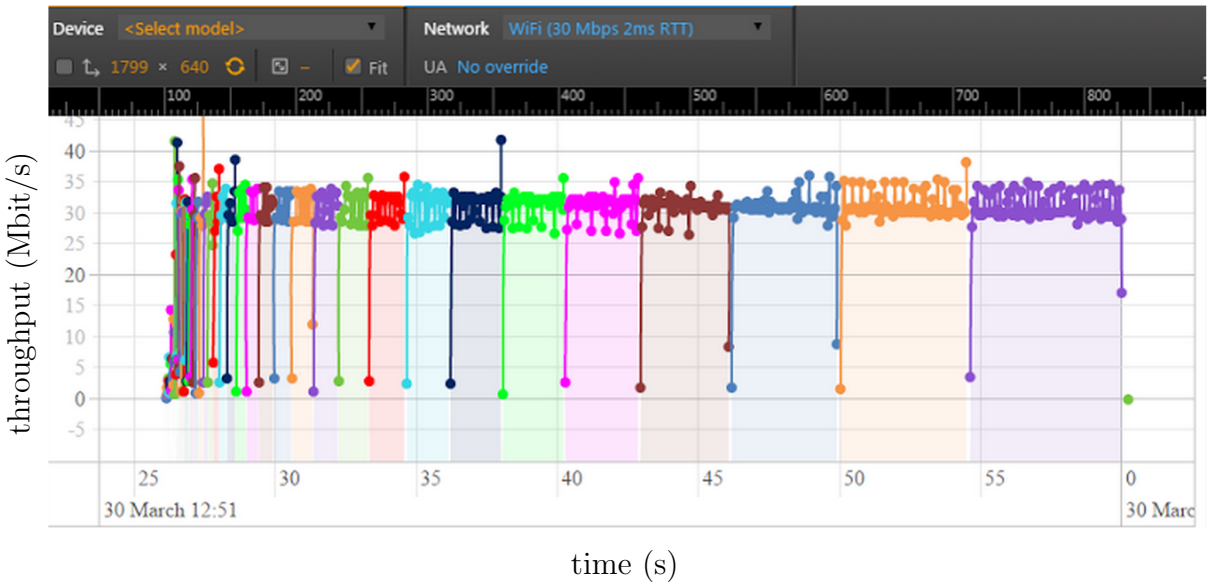


Figure 3.10: The throughput for each TCP session during a TCP download test with a setting of one TCP session. The test is conducted when Google Chrome keeps the throughput under 30 Mbit/s with 2 ms RTT.

3.3.3 Websockets

The throughput calculation was also done on the browser. But the websocket API only supported reporting the throughput at some arbitrary time. The aggregation of the throughput for many simultaneous websocket streams was thus not trivial. The approximation of the aggregated throughput is presented here.

Figure 3.11 shows two streams that report throughput at different timestamps. T should be picked so at least of the streams don't need approximation when aggregating the streams. The remaining streams will be interpolated at point T by the two point formula for a straight line

$$y - y1 = \frac{y2 - y1}{x2 - x1} \cdot (T - x1) \tag{3.1}$$

where y is the transferred amount of data we seek for one session. The aggregation is complete by adding up the results when all sessions have been interpolated.

3.3.4 TCP Session

The measurement of the metrics defined in RFC 6349 was implemented in big part by investigating the TCP session through TCP_INFO. Note that both AJAX and Websockets use TCP, by implementing the measurement of these metrics on the TCP layer they can be measured for both the AJAX and Websocket technologies.

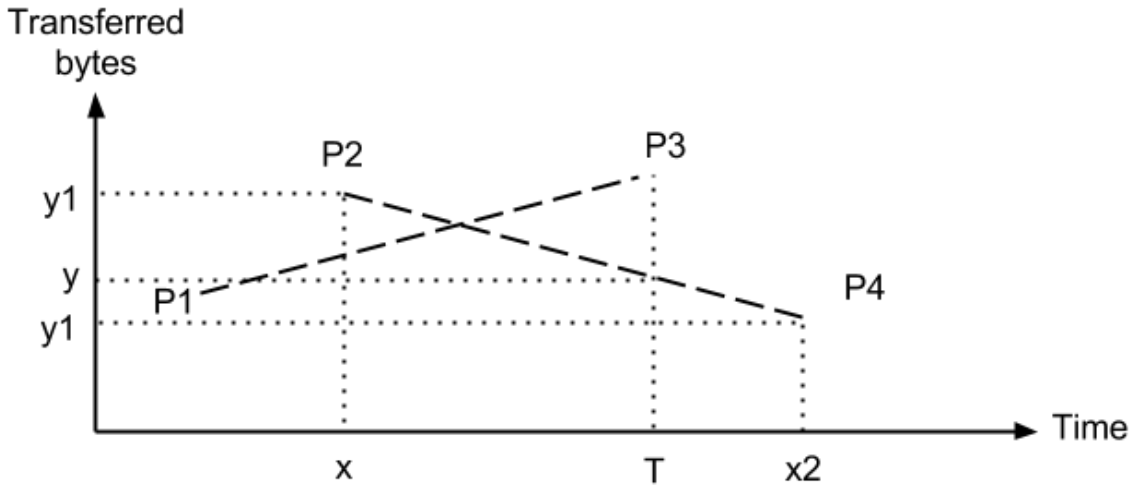


Figure 3.11: Multiple streams that report timestamp and the amount of data transferred at different intervals. P1 and P3 belongs to one TCP session, P2 and P4 belongs to another TCP session.

Baseline Round Trip Time

RTT was measured by having the server send a response to the client and wait for the tcpi_unacked variable to be zero so it is known the message have been acknowledged by the client. The round-trip time RTT will then be retrieved by calculating the difference between tcpi_last_data_sent and tcpi_last_data_recv. This strategy places the calculations and sampling in the kernel and the assumption is that it should be accurate for this reason. This is particularly relevant since TCP is a stream oriented protocol rather then a packet oriented protocol, meaning packets are not sent immediately when the user space sends data on a TCP socket. But it would be possible to measure accurately on application layer if the test had a dedicated TCP stream where data is inserted to the stream at high intervals. The client side would have to attempt to reflect the data as soon as possible. The server can then compare the time it inserted the data to when it got the packet back from the reflection. The problem here is that it would also measure the delays in the application and TCP stack, not only the network delay. It would be another story if both network cards of the server and client were exposed to the application since the application then would be able to ask the network card for the time a packet was received. But only the servers network card is exposed to the application in this thesis. Another approach that would work for the baseline test would be to implement the entire TCP stack that can use the TCP ACKs instead of relying on the applications data reflection. And since the servers network card can be used the measurements could be very accurate. But it was decided to go with the simple approach of using the kernel since it could be

as good, require very little implementation and when considering the scope of this thesis. It would also be excessive to compare a very accurate baseling RTT to the RTT during a throughput test that never can be so accurate. Yet another solution that could work for the baseline test would be the HTTP status code "302 Found" that instructs the browser to make a new request to a new page. This way the server could compare the timestamps between the 302 request and the new request on the new page. It would however include the delay of the browsers that could be different depending on browser implementation. The biggest problem with the chosen solution was that it relied on the assumption that the TCP ACK messages are sent as soon as possible. This is not always the case as TCP is designed to ACK in bulks during high throughput. But for very low throughput it should send the ACK messages immediately.

Round Trip Time During Load

The TCP_INFO exposes tcpi_rtt and tcpi_rcv_rtt that is updated by the kernel with the sender and receiver TCP socket respectively. These were first chosen for measuring the RTT during load for the same reason TCP_INFO was choosen when measuring the baseline RTT. Namely because of the assumption that it should be accurate since the kernel calculates these metrics and because the scope of this thesis cannot afford investing time into rewriting the TCP stack. The alternatives are covered in detail in the above section when discussing the baseline RTT.

Jitter

The TCP_INFO metric tcpi_rttvar exposes the RTT variation estimation the kernel performs for TCP when sending packets. One can argue this TCP_INFO metric should be used for measuring the jitter for the same reasons TCP_INFO metrics for the rtt was selected. But TCP have inherent jitter as explained in the theory chapter so those accuracy arguments does therefor not hold. Another problem with tcpi_rttvar is that it only updates for the down direction since the variable only was updated by the kernel when it sent packets. See figure 3.12 for a insight to this variable. The method selected for obtaining the jitter over TCP was to compute the difference between min and max RTT delay. This was explained in the theory chapter.

Transfer Time Ratio

There is little debate on how this metric would be calculated as it is defined in a RFC. The user will input the link speed, LS, as it cannot be known by the test. The transfer time ratio metric, TTR, was then calculated by using the Ideal TCP Transfer Time

$$ITT = \frac{B}{(tcpi\_pmtu - 40) \cdot 8 \cdot LS/(P \cdot 8)} \tag{3.2}$$

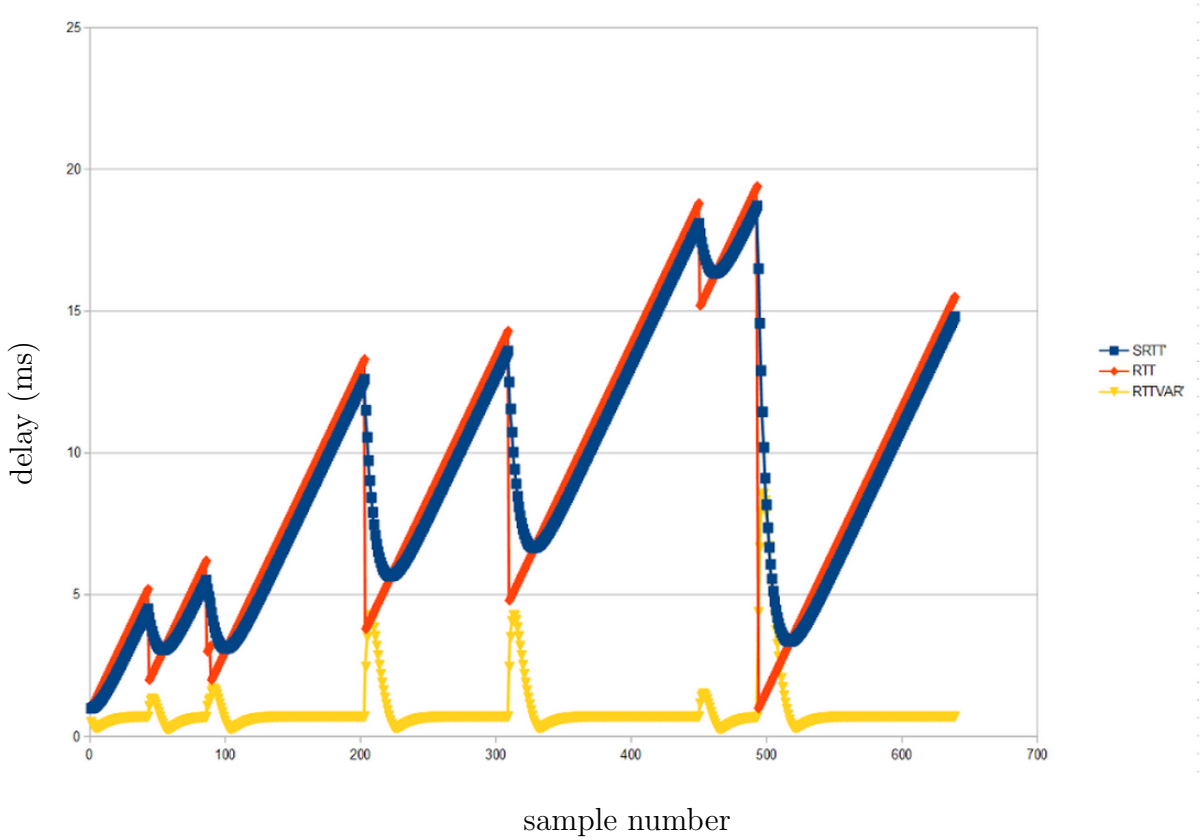


Figure 3.12: Smoothed RTT and RTT variation calculated by their formula by using 650 samples of fabricated RTT values that increase by one before suddenly dropping. The formula can be found in table 2.1.

where the transferred bits, B, was trivially acquired by counting when the throughput is no longer affected by auto-tuning. The RFC derived different physical packet sizes, P. 1538 bytes was derived for IP version 4, which was used in these tests. This would not be have been correct if the network was using the 802.1q standard, where P would have been 1542 instead of 1538 [9]. This difference is small and neglected for this prototype. How it is decided that the auto-tuning phase is complete could be done in many ways. Analyzing the throughput over time maybe would be possible. Tracking the TCP_INFO congestion state maybe would give repeatable results. Approximating a timeout based on historical data from tests with many browsers and platforms maybe would work. Maybe all of the above together would yield the best result. Considering the scope of this thesis were a simple and arbitrary time chosen that was considered the time it took for the auto-tuning phase to be complete.

Finally was the transfer time ratio metric

$$TTR = T/IT \tag{3.3}$$

where the actual throughput T was calculated during the transfer and the ideal throughput IT was known.

TCP Efficiency

Each packet should be the size of the path MTU without the TCP header, assuming every packet is filled before it is sent. Thus is the TCP efficiency

$$E = 100 \cdot \frac{B - tcpi\_total\_retrans \cdot (tcpi\_pmtu - 40)}{B} \tag{3.4}$$

where the number of transmitted bytes, B, extracted trivially from the TCP stream. The header of each packet was 40 bytes. During the auto-tuning phase were the transmitted bytes not counted. The same auto-tuning phase derivation were done for this metric as the previous.

Buffer Delay

The calculation of this metric was calculated as defined in the RFC. The inputs to the formula is the RTT baseline and the average RTT. The RTT baseline was handled as an input, could be from an external test or simply user input. The tcpi_rtt variable was used for calculating the average RTT during a test. It does not represent the average RTT for the connection, but it might provide a good approximation of the actual average RTT by sampling the smoothed RTT variable a few times per second and calculating an average value. An alternative approach of obtaining the average RTT during a throughput test would be to run the Baseline test parallel with the throughput test. The buffer delay could in that case be more accurate as it probably would not include the server and clients buffers. It would however require a dedicated connection for the baseline test, something that is in short supply in the browser and could be devastating for the throughput test as it needs to saturate a gigabit connection.

Receive and Send Buffers

The server's send and receive buffers was easily retrieved by using the socket's API function getsockopt that was exposed in the C programming language. The browser's receive buffer was approximated by continuously sampling the tcpi_rcv_buff variable during the throughout test and finding the maximum value of those samples. When using multiple TCP sessions were they sampled at the same time and aggregated before compared with the previous sample. The same approach was taken when approximating the browser's send buffer but instead was the tcpi_unacked variable used. Additionally was the value multiplied by the path MTU since the tcpi_unacked is measured by packet number.

Increase Receive and Send Buffers

For a fully RFC 6349 compatible test must the receive and send buffers not bottleneck the throughput [9]. It is not possible to increase these buffers on the browser side without changing global settings on the machine or the browser.

The receive and send buffers was raised on the server to huge limits so the servers send and receive buffers would not be a bottleneck for the throughput. The files /proc/sys/net/-core/rmem_max and /proc/sys/net/core/wmem_max were edited from 212992 to 1277952 (6 times bigger) in order to increase the buffers beyond 212992 bytes. This allowed the send and receive buffers to go just over 1MB, which would allow for a throughput of

$$T = \frac{RWND \cdot 8}{RTT} = \frac{1277952 \cdot 8}{10} = 1022.4Mbit/s \tag{3.5}$$

for a single TCP session when using a RTT of 10 ms. Note that the browser and its machine's buffers were not changed as it would be impossible in the use case posed by this thesis.

3.3.5 WebRTC

Before investing time into implementing the server side WebRTC test were a small browser to browser proof of concept quickly developed. This was shown to save lots of time as the browser mysteriously crashed when running the test. Why it cashed is presented in the next chapter. The same crash would have happened if the test were implemented on the server. No more time was invested into the WebRTC technology for this reason. Because the datachannel protocol is packet oriented instead of stream oriented the implementation of the metric measurements were strait forward and handled in the application layer in JavaScript. The datachannel were configured to be unreliable so loss could be detected. The packet loss information was obtained by maintaining a buffer that counted the sequence numbers that were artificially inserted to the payload. The inter-arrival jitter was calculated by comparing the timestamps in the payload to the current time of the receiver. The loss distance and loss period should also easily be obtainable once the assumption that loss is measurable with Data Channels is verified. Note however that the delay, jitter and loss metrics are measured separately for each direction. For accurate delay and jitter measurements must the clocks of the machines be in sync. This problem were never fully solved in this thesis since the browser crashed anyway. Further investigation would include to try a implementation of NTP **??** that is meant to run in the browser. Alternatively could the measurements instead be implemented so it reflects the packets and thus only measure jitter and delay based on round trip. The packet loss could still be measured for both directions separately.

3.3.6 Credits

The ambition was to implement all of the defined tests, including the Quality Test that needed to be implemented with WebRTC's datachannels on the server side. For this reason was the web server implemented by extending a example code snippet in the WebRTC native code package [32]. The only other alternative found was OpenWebRTC, but it did not support datachannels [33]. Since the example code snippet and the code package was written in C would the server prototype also be written in C.

The example code snippet came with a simple HTTP parser that ended up being used by the prototype. The example was not threaded and needed heavily modifications. The Websocket parser that was used is called WebSocket [34], it was chosen for its simplicity.

3.4 Prototype evaluation

The final phase in the thesis were to evaluate how the prototype performed. This was done by introducing different network scenarios and compare the measurements with other tools. These other tools did not run in the browser. The scenarios were chosen based on real world examples. 20 ms delay was for example one of the scenarios because it is a typical delay when communicating within Sweden. The supervisors at Netrounds gave good knowledge of such scenarios and were the main source of such examples.

3.4.1 Test Environment

Testing of the prototype on a real network came natural since all necessary components already existed after the implementation phase. See figure 3.13 for a view of the network topology. The server had the prototype installed and the browsers Chrome and Firefox that were used when evaluating the browser to browser proof of concept. The Windows 7 machine had Opera and Internet Explorer additionally installed. The switch is also connected to the office network that among other services had a DHCP server running. It is vital that the noise of such a network does not interfere with the test, this was the reason for having the office network connected during the test.

It was verified that the the network between the two machines could saturate a gigabit connection in both directions. This was verified by a known test tool called iperf [35], see figure 3.14.

Network emulation tools

The Linux machine had the netem [30] tool installed for emulating delay, packet loss and jitter in the network. The Windows machine could also emulate changes in the network using a tool called Clumsy [36].

Netem is a network emulation tool that manipulates outgoing traffic on the interface. It is quite trivial how to generate packet loss and a steady delay. But when generating

jitter can reordering be introduced if not extra steps is taken. Reordering can be counted as packet loss under some definitions. In this thesis is reordering not counted as packet loss and reordering is not measured, for this reason were the extra steps not taken to combat reordering. When the results is presented in the next chapter is also the used netem command presented.

Clumsy is similar to the netem tool but netem is for Linux and Clumsy is for Windows. Netem is infinitely more powerful and complex while Clumsy only have a few options. Clumsy was used in this thesis for emulating loss and delay.

It would have been preferred if the same network emulation tool would have been used on both machines. It would mean less variables and thus more trusted results.But this simply was not possible with the available hardware. One additional machine in between the windows and server machine would have been needed that would have had two interfaces that the traffic would have travelled trough. This way could netem been able to manipulate the traffic in both directions the same way. And the network manipulation would have been symmetric. Most importantly jitter would have been able to be generated in both directions.

Google Chrome Developer tools

The Google Chrome browser have a powerful tool for developers that can measure network timings, throughput, RTT and more. It can even throttle the throughput. This tool was used during the evaluation phase as a reference for the RTT, throughput and number of concurrent streams.

3.4.2 Considered alternatives

Most of the metrics the prototypes measures were evaluated with little complication and were strait forward. But for some metrics were extra consideration required. These extra considerations is mentioned here.

Path MTU

Because the Path MTU is the minimum MTU of the entire path of the network there is countless of scenarios with corner cases that can be included in the test network. A scenario that considers path MTU would be a minimum of four computers connected in series where the two computer in the middle functions as routers. Each two connected interfaces that makes up a link would have different MTU sizes. See figure 3.15 for such a scenario. With a scenario like this the packets would be fragmented by the network for both directions. Note however that this thesis does not cover this case due to the limited scope of this thesis and lack of hardware. Thus making this a item of improvement when discussion further work.

Metric precision

The emulated delay, jitter and loss have a error margin on the real network that inhibits conclusions on the precision of the measurement. One alternative to combat this would be to measure the same traffic with a tool that is established to have high precision. The traffic can even be recorded for calculating the metrics offline. Another alternative could be to simulate the traffic by playing a file of known traffic that already have the metrics calculated. But that would not be feasible when dealing with TCP and SCTP streams.

But the ultimate use case, to troubleshooting a connection with the browser, does not require high precision of the metrics. Meaning that it is acceptable if setting a delay of 20 ms actually generates a delay of 22 ms. Little effort were invested in other approaches for this reason.

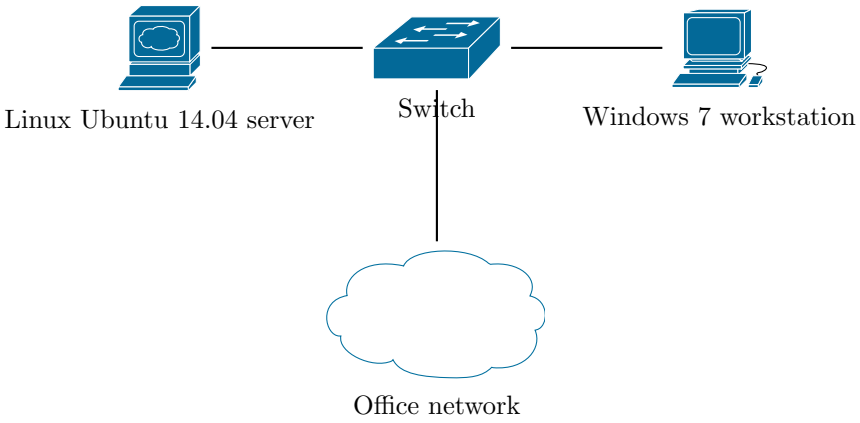


Figure 3.13: Network topology that was used during the evaluation phase.

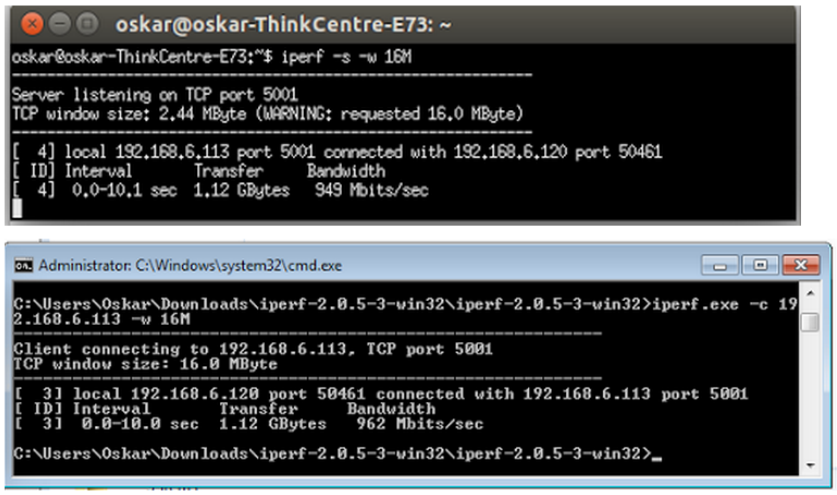


Figure 3.14: Iperf's measurement of throughput in both directions (upload and download). Note the receive and send buffers are well beyond the bottleneck limits.

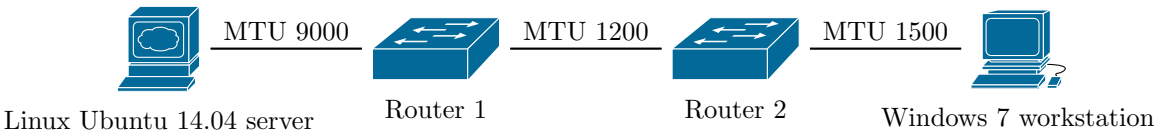


Figure 3.15: Alternative network topology that considers path MTU.

Chapter 4

Results

This chapter presents the results of the implemented prototype. The WebRTC Data Channels metrics is presented using a separately implemented proof of concept.

4.1 AJAX Throughput Performance

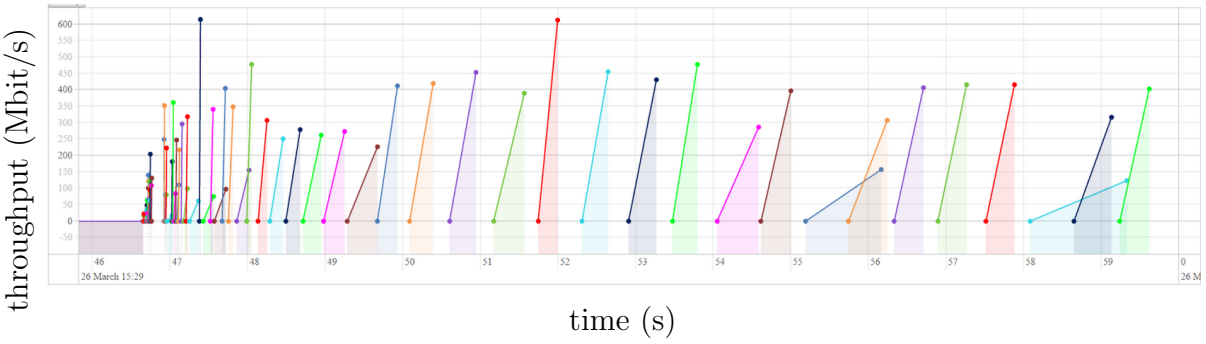


Figure 4.1: The throughput for each TCP session during a TCP upload test with a setting of five parallel TCP sessions. The test was conducted on a network with minimal delay.

The root cause of the performance problem of the browser for AJAX throughput can be captured by comparing figures 4.1 and 4.2. The settings for the tests are five parallel streams but with 10 ms is the actual value around three. For minimal delay is it usually only one active parallel TCP session. The parameters that effect this scenario are the delay between the server and client, the file-size the connection is download/uploading and the link speed. Since the link speed is defined to 1000 Mbit/s in the goals of this thesis is only the delay and file-size considered. And the test is quite useless if it cannot support a delay of 10 ms. That leaves the file-size. Unfortunately does the browser crash

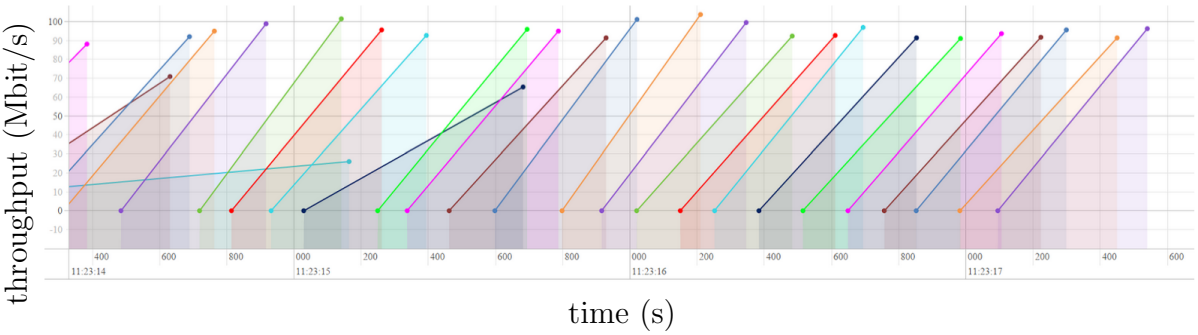


Figure 4.2: The throughput for each TCP session during a TCP upload test with a setting of five parallel TCP sessions. The test was conducted on a network with 10 ms RTT delay.

for a upload test if the file-size is bigger then 54 MB. So the AJAX technology does not fit the throughput test defined in this thesis.

4.2 Websockets Throughput Performance

The websocket thechnology performs quite well as seen in table 4.1, where the number of channels were increased until the throughput did not increase. The upload test does however not perform as good as the download test because the browser is generating and sending the data to the server at the Application layer in JavaScript. It simply is not efficient enough. Internet Explorer did also perform bad due to its limit of 6 simultaneous datachannels.

4.3 WebRTC Performance

The WebRTC tests were conducted with a small browser to browser, proof of concept, JavaScript application.

4.3.1 Throughput

The messages received is always 300/s until the size of the messages is over 25 kB (25 000 bytes). So the maximum throughput measured by the proof of concept is 60 Mbit/s. The throughput suffers immensly if the size of the message is over 25 kB. The throughput also suffer greatly if there is delay in the network. When running the test bidirectionally the is throughput halved and when using more data channels is the combined throughput the same. There is no difference in any combination of browser or platform. See talbe 4.2 for a presentation of the throughput numbers for the mentioned scenarios. The results are very poor when considering the goal of a throughput of 1000 Mbit/s.

Browser	RTT	Direction	Channels	Throughput
Chrome 40.0	10 ms	up	8	773 Mbit/s
		down	5	917 Mbit/s
	20 ms	up	20	792 Mbit/s
		down	10	510 Mbit/s
Firefox 37.0.2	10 ms	up	15	943 Mbit/s
		down	5	939 Mbit/s
	20 ms	up	20	946 Mbit/s
		down	20	570 Mbit/s
IE 11	10 ms	up	6	115 Mbit/s
		down	5	949 Mbit/s
	20 ms	up	6	58.8 Mbit/s
		down	6	572 Mbit/s
Opera 29.0	10 ms	up	20	830 Mbit/s
		down	6	935 Mbit/s
	20 ms	up	25	880 Mbit/s
		down	7	580 Mbit/s

Table 4.1: The summary of throughput tests for each direction with different delays and browsers.

Browser combination	Delay	Direction	Channels number	Throughput
Chrome 40.0 & Chrome 40.0, Firefox 37.0.2 & Firefox 37.0.2 or Chrome 40.0 & Firefox 37.0.2	0 ms	single	any	60 Mbit/s
		bidirectional	any	30 Mbit/s
	10 ms	single	any	5 - 10 Mbit/s
		bidirectional	any	2 - 5 Mbit/s

Table 4.2: The summary of throughput tests of the WebRTC data channel with different browsers, delays, direction and number of data channels. The throughput is measured in only one direction, meaning when there is a bidirectional test the presented throughput is only the throughput for one direction.

4.3.2 RTT and Jitter

The delay and jitter is only measured correctly over the application layer if the message size that is feed into the data channel's send function is under the configured MTU. See table 4.3 for a summary of the performed tests that lead to this concluation. The netem command for achiving delay and jitter was "tc qdisc change dev eth0 root netem delay 100ms 10ms distribution normal". Note that the delay is very sporadic but that the inter-arrival jitter is 1 ms. This can happen if the delay walks up and down with a step of 1 ms. Also consider the impact on the throughput with this message size, it would yield only 3.4 Mbit/s since the transmission rate is 300 messages/s.

Browser combination	Message size	MTU	Delay	Jitter	Measured delay	Measured jitter
Chrome 40.0 & Chrome 40.0, Firefox 37.0.2 & Firefox 37.0.2 or Chrome 40.0 & Firefox 37.0.2	25 000 bytes	1500 bytes	0 ms	0 ms	2 ms	0 ms
			10 ms	0 ms	100 - 300 ms	1 ms
	10 000 bytes	1500 bytes	10 ms	0 ms	100 - 300 ms	1 ms
		11 000 bytes	100 ms	10 ms	103 ms	10 ms
	1400 bytes	1500 bytes	10 ms	0 ms	13 ms	0 ms
			100 ms	10 ms	103 ms	10 ms

Table 4.3: This is a summary of delay and jitter measurements over the application layer with WebRTC data channel.

4.3.3 Packet Loss and out of order

When loss is introduced with clumsy or netem does the browser crash immediately. Same results with all combinations of platforms and browsers. The Firefox crash report can be traced to a function named sctp_handle_sack [37]. When reordering is introducing on the outgoing interface with either the clumsy or netem tool is the packets reordered also at application layer without any complications.

4.4 TCP Metric Precision

The metrics under this section is relevant for websockets and AJAX. Much of the TCP metrics is relying on assumptions about the TCP_INFO struct. Most of these TCP metrics are verified by verifying these assumptions.

4.4.1 RTT Baseline

The real delay is set to 5 ms with netem and clumsy on respective machines and the RTT is verified to be 11 ms with the ping command line tool. During these conditions does the RTT baseline test of the prototype measure a RTT of 12, 12, 8, 8, 12, 8... Further investigation shows that the precision of tcpi_last_data_sent and tcpi_last_data_recv seem to be $\pm$ 2 ms since the variables only updated by a multiple of 4 (0, 4, 8, 12, ...). The average of many RTT Baseline tests still seam to converge close to the true value. See table 4.4 for the results of the performed baseline tests.

4.4.2 Buffer delay

The buffer delay is depending on the user input of the RTT baseline and two variables from TCP_INFO for the different directions. The smoothed RTT variable tcpi_rtt is measured during a download throughput test. The receiver estimated RTT variable tcpi_rcv_rtt is measured during a upload throughput test. By verifying these TCP_INFO variable is the buffer delay verified. The real delay is set with netem and clumsy on

Duration	Samples	RTT	RTT samples	RTT samples average
20 s	40	8 ms	8, 8, 8, 8 ... ms	8.0 ms
		10 ms	12, 12, 8, 8, 12, 8... ms	10.8 ms
		11 ms	12, 8, 12, 12, 12, 8... ms	11.3 ms
		20 ms	20, 20, 20, 20, ... ms	20.0 ms
		21 ms	20, 20, 24, 20, 24 .. ms	22.6 ms
		100 ms	100, 100, 100 ... ms	100.0 ms
		102 ms	100, 100, 104, 104, 100... ms	104.4 ms

Table 4.4: This is a summary of the evaluation of the RTT baseline test.

Duration	Samples	Direction	Variable	RTT	Measured RTT average
60 s	60	up	tcpi_rcv_rtt	11 ms	11.6 ms
				20 ms	21.3 ms
				50 ms	48.4 ms
		down	tcpi_rtt	11 ms	11.2 ms
				20 ms	22.1 ms
				50 ms	52.8 ms

Table 4.5: This is a summary of the evaluation of the tcpi_rtt and tcpi_rcv_rtt variable from TCP_INFO.

respective machines and the RTT is verified with the ping command line tool. During these conditions does the upload and download throughput test report the RTT from their respective TCP_INFO variables. In table 4.5 can the results of the evaluation of this metric be found.

4.4.3 TCP Jitter

The TCP Jitter is the difference between min and max delay values. Thus this calculation is evaluated against different network scenarios created by the netem tool. Note that the jitter cannot be manipulated by clumsy. Meaning the jitter therefor is not applied on outgoing packets from the client. The jitter is asymmetric and only in the down direction. But the prototype should despite this be capable of measuring the jitter in the up direction since the jitter is calculated based on RTT delay (note the Round Trip in RTT).

Three different tests were run where the first had no netem constraints, the second had the netem command "tc qdisc change dev eth0 root netem delay 100ms 10ms distribution normal" and the third had the netem command "tc qdisc change dev eth0 root netem

Duration	Samples	Direction	RTT	Jitter	Measured RTT	Measured jitter
60 s	60	down	0 ms	0 ms	1.2 ms	0.2 ms
			100 ms	10 ms	101.3 ms	12.1 ms
			100 ms	50 ms	98.6 ms	61.2 ms
		up	0 ms	0 ms	2.4 ms	0.7 ms
			100 ms	10 ms	102.7 ms	13.6 ms
			100 ms	50 ms	104.2 ms	59.8 ms

Table 4.6: This is a summary of the evaluation of the tcpi_rcv_rtt and tcpi_rtt for up and down directions respectively for RTT. These variables are from TCP_INFO. Jitter is measured with difference between min and max RTT delay.

Direction	Packet loss	Measured TCP efficiency
down	0 %	100 %
	1 %	99.05 %
	5 %	94.85 %

Table 4.7: The samples from a download test when netem is configured with packet loss.

delay 100ms 50ms distribution normal". This translates to 0, 10 ms and 50 ms jitter and was verified by comparing against the mdev variable in the traditional ping tool. See table 4.6 for all the results and parameters of the tree tests.

4.4.4 TCP Efficiency

This metric is derived from the tcpi_total_retrans variable. The real packet loss probability was set by netem that runs on the same machine as the prototype. Three different tests were run where the first had no netem constraints, the second had the netem command "tc qdisc change dev eth0 root netem loss 1%" and the third had the netem command "tc qdisc change dev eth0 root netem loss 5%". This translates to 0% packet loss, 1% packet loss and 5% packet loss. See table 4.7 for the results.

Wireshark were used for verifying the assumption that packets are the same size as the path MTU.

4.4.5 Path MTU

Both the Linux and Windows machines can easily change the MTU for an interface. Lowering the MTU on either interface should yield a lower the path MTU. A few values were tested from 1500 to 552. All tested MTU's changed the path MTU to the correct value. See table 4.8 for the results from the tests.

Server MTU	Client MTU	Measured Path MTU
1500	1500	1500
1500	552	552
552	1500	552
743	743	743

Table 4.8: The tests performed for evaluating the path MTU.

Bytes transferred	Duration	Measured bytes transferred	Measured duration
1168 Mbit	5.79 s	1166 Mbit	5.81 s

Table 4.9: The tests performed for evaluating the transfer time ratio.

4.4.6 Transfer Time Ratio

This metric is using the path MTU, the bytes transferred, the duration of the transfer and input of the Bottleneck Bandwidth metrics. The transfer time ratio is verified by verifying these metrics. The Bottleneck Bandwidth verification is trivial since it is a user input. The verification of bytes transferred and the duration of the transfer was verified with the Google Chrome debugger reports in a simple test.

During a short test was 1168 Mbit over 5.79 seconds reported by the prototype, when the browser measured 1166 Mbit over 5.81 seconds. See table 4.9.

4.4.7 DSCP

When inputting something to the DSCP in a TCP download test the output DSCP from the test should equal the input. The decimal number 63 was used for testing all 6 of the DSCP bits, which translates to 111111 in binary. The default value is 000000. Wireshark was used for capturing the outgoing packets from the server and verifying that the DSCP value actually is transmitted over the network.

The packets captured with Wireshark had all the DSCP bits set to 1 when the DSCP input from the test was 63. The output of the test had the same DSCP value.

4.4.8 Receive and Send Buffers

The browsers receive and send buffers in both directions depend on tcpi_rcv_space and tcpi_unacked. This metric can be verified when these variables are verified to be accurate. The actual throughput during a test should be close to the calculated achievable throughput when derived from these metrics and when the TCP efficiency is 100%.

Seven tests in total were run with different delays, 100% TCP efficienct and with only one TCP session.

The conclusion can be drawn that tcpi_unacked gives a good approximation when analyzing the tests in table 4.10. And that tcpi_rcv_space have a correlation with the throughput that easily can be mapped to the actual value.

Direction	RTT	Throughput	tcpi_unacked based throughput	tcpi_rcv_space based throughput
Upload	50 ms	22 Mbit/s	-	41 Mbit/s
	20 ms	54 Mbit/s	-	80 Mbit/s
	10 ms	105 Mbit/s	-	220 Mbit/s
	5 ms	207 Mbit/s	-	400 Mbit/s
Download	20 ms	104 Mbit/s	102 Mbit/s	-
	10 ms	204 Mbit/s	203 Mbit/s	-
	5 ms	395 Mbit/s	392 Mbit/s	-

Table 4.10: Summary of 7 tests where RTT and direction were alternated. Only one TCP session were used and no packet loss recorded.

CHAPTER 5

Discussion and conclusion

The goals of the thesis have been met. Tests have been presented that utilizes the modern web browsers capabilities, without third party software. Most of these tests have been implemented in a prototype and evaluated. The remaining tests, where the WebRTC Data Channel is involved, was evaluated using a quick proof of concept implementation that yielded disappointing results. The TCP tests can saturate a gigabit connection with a web browser.

The presented TCP Throughput Test implements many of the metrics defined in RFC 6349. This can give engineers deeper understanding into why the network behaves as it does. The results of the TCP Throughput Test is accurate. The browsers send and receive buffers are approximated only during low packet loss, which should be acceptable since the packet loss probably then is the bottleneck.

Obtaining maximum throughput over a gigabit line is possible via the browser using AJAX when downloading from the server. It is also possible with Websockets when downloading or uploading. Not all browsers can saturate a gigabit line, the best performing browser was Firefox.

A test have been presented for verifying the user's QoS settings by using the DSCP value or the connecting port as a marker. Network engineers can with this test compare different RFC 6349 reports of TCP sessions with different QoS markers for validating the networks QoS capabilities.

WebRTC Data Channels is currently not ready for packet loss statistics when the browser sends data. But it should still be possible to calculate the jitter and delay with reasonable precision. The highest throughput for Data Chennel was 60 Mbit/s between two browsers. It may be possible to improve performance by measuring between a native WebRTC client and one browser instead of between two browsers. These conclusions might be explain why no related work was found that measures network quality with this technology.

The contributions from this theses have shown that the traditional speed test websites can include more network measurements. These additional metrics can save time when

troubleshooting network connections. The metrics is also proven to be accurate except for the throughput for high delay. The similar but basic solution Bredbandskollen that was mentioned in the related work section gave a download throughput higher then theoretical max, maybe that is an effect from them trying to compensate for the high delay. The noted shortcomings of the similar solution NDT have been addressed during this thesis. It is extremely clear what metric belong to the different tests, each metric is thought out and have good motivation for being presented to the human user and they are defined in RFC's instead of by the Linux kernel. This thesis even hare metrics that NDT does not have even thou NDT have almost a hundred metrics. MTU is for example a metric NDT does not have [4].

5.1 Further Work

This thesis was held within a limited time frame and thus have some planned work been demarcated for this chapter.

5.1.1 WebRTC

The video and audio part of WebRTC have not been implemented or evaluated for usage as network diagnostics. It would need a native WebRTC client for extracting the interesting RTCP statistics. The planed native WebRTC client have not been implemented because the WebRTC library introduced time consuming problems. This client should be implemented and evaluated before it can be included in the presented tests UDP Upload Quality and UDP Download Quality.

5.1.2 Dynamic TCP Sessions

The number of simultaneous TCP sessions is currently implemented as a input to the test. This number should be dynamically configurable for maximizing the throughput, independent of web browser. The test can use the feedback from the server for evaluating if more sessions should be opened. Specifically should it look at the maximum throughput allowed by the browsers buffer size.

5.1.3 Baseline RTT Accuracy

The accuracy of the RTT Baseline Test was not very good. A alternate solution, without using tcpi_last_data_sent and tcpi_last_data_recv, is to use a HTTP redirect. The HTTP redirect can instruct the browser to instantly make another request. The server can calculate the RTT by comparing the timestamp from when it sends a HTTP redirect to the client and the timestamp from when the client responds to that HTTP redirect.

5.1.4 Test MTU Metric

The test environment of only two computers and the limited scope of this thesis did not allow the MTU metric to be fully evaluated during more complex network scenarios. The minimum test environment required is described in 3.15.

5.1.5 Combining AJAX and Websockets

For browsers that cannot saturate a gigabit connection with websockets alone maybe can do when combining Websockets with AJAX. This may solve the problem for browsers with a limit on the number of websocket connections and when they are dealing with high delay. This is only a viable solution when the browser's buffers are the limiting factor. There is little that can be done if the CPU on the client is a bottleneck.

5.1.6 802.1q Standard Support

When calculating the Maximum Achievable Throughput used in the Transfer time ratio was the networks with the 802.1q standard neglected. IP version 6 was also neglected. It can be implemented simply by more user input, or by using detection on the server side by parsing a single packet using raw sockets. This improvement would effect even more metrics's accuracy since many relay on a TCP packet header of 40 bytes.

5.1.7 Evaluate More Platforms

Only Windows 7 was tested in this thesis. Not only should Linux and Mac be evaluated, but also handheld platforms. Windows machines usually have the third party software installed while it is the other platforms that cannot rely on Flash or Java applets. This is a strong reason for evaluating the other platforms.

5.1.8 Web Workers

The test may be improved for browsers that did not saturate a gigabit connection by using web workers. Web workers are for improving the JavaScript performance. This technology is supported by all modern browsers.

APPENDIX A

TCP_INFO Derived Variables

The documentation of the TCP_INFO variables is bad. Most variables was defined by experimentation or reading the source code. A full list of the TCP_INFO struct is visable in table A.1.I have derived the description on my own.

TCP_info variable	Description
tcpi_state	TCP state TCP_ESTABLISHED, TCP_SYN_SENT, etc
tcpi_ca_state	Congestion control state (slow start, avoidance, fast recovery)
tcpi_probes	-
tcpi_snd_wscale	Sending window scale, number of shifts.
tcpi_rcv_wscale	Receiving window scale, number of shifts
tcpi_rto	Retransmission timeouts. Not accumulating.
tcpi_ato	Timeout period for sending a delayed ACK. Predicted tick of soft clock
tcpi_snd_mss	Cached effective maximum segment size
tcpi_rcv_mss	MSS used for delayed ACK decisions
tcpi_sacked	Number of selective ack:s. Not accumulating.
tcpi_lost	Lost packets. Not accumulating.
tcpi_retrans	Number of non-timeout based retransmits. Not accumulating.
tcpi_fackets	Number of FACK (forward ACK) packets. Not accumulating.
tcpi_advmss	Advertising maximum segment size
tcpi_reordering	Setting of how much a packet may be reordered before dropped
tcpi_unacked	Number of not ack:ed transmits. Number of packet "in flight".
tcpi_last_data_sent	Timestamp delta when last data was sent
tcpi_last_ack_sent	Not implemented
tcpi_last_data_recv	Timestamp delta when last data was received
tcpi_last_ack_recv	Timestamp delta when last ack was received
tcpi_pmtu	Path maximum transmission unit
tcpi_rtt	Smoothed round trip time [29] $SRTT = RTT$ $SRTT' = (7/8) \cdot SRTT + 1/8 \cdot RTT'$
tcpi_rttvar	Round-trip time variation [29] $RTTVAR = RTT/2$ $RTTVAR' = (3/4) \cdot RTTV + 1/4 \cdot \lvert SRTT - RTT' \rvert$
tcpi_rcv_ssthresh	Receiving slow start size threshold
tcpi_snd_ssthresh	Senders slow start size threshold
tcpi_snd_cwnd	Senders congestion window size
tcpi_rcv_rtt	Receivers RTT estimation
tcpi_rcv_space	Receiver queue space
tcpi_total_retrans	Retransmission for the entire connection

Table A.1: Some of the variables in the TCP_INFO struct.

Appendix B

Prototype Output Example

The user experience is not within this thesis's scope. Therefor is the screenshots limited to this appendix. Note that some feedback to the user is continuous through the test. See figure B.1.

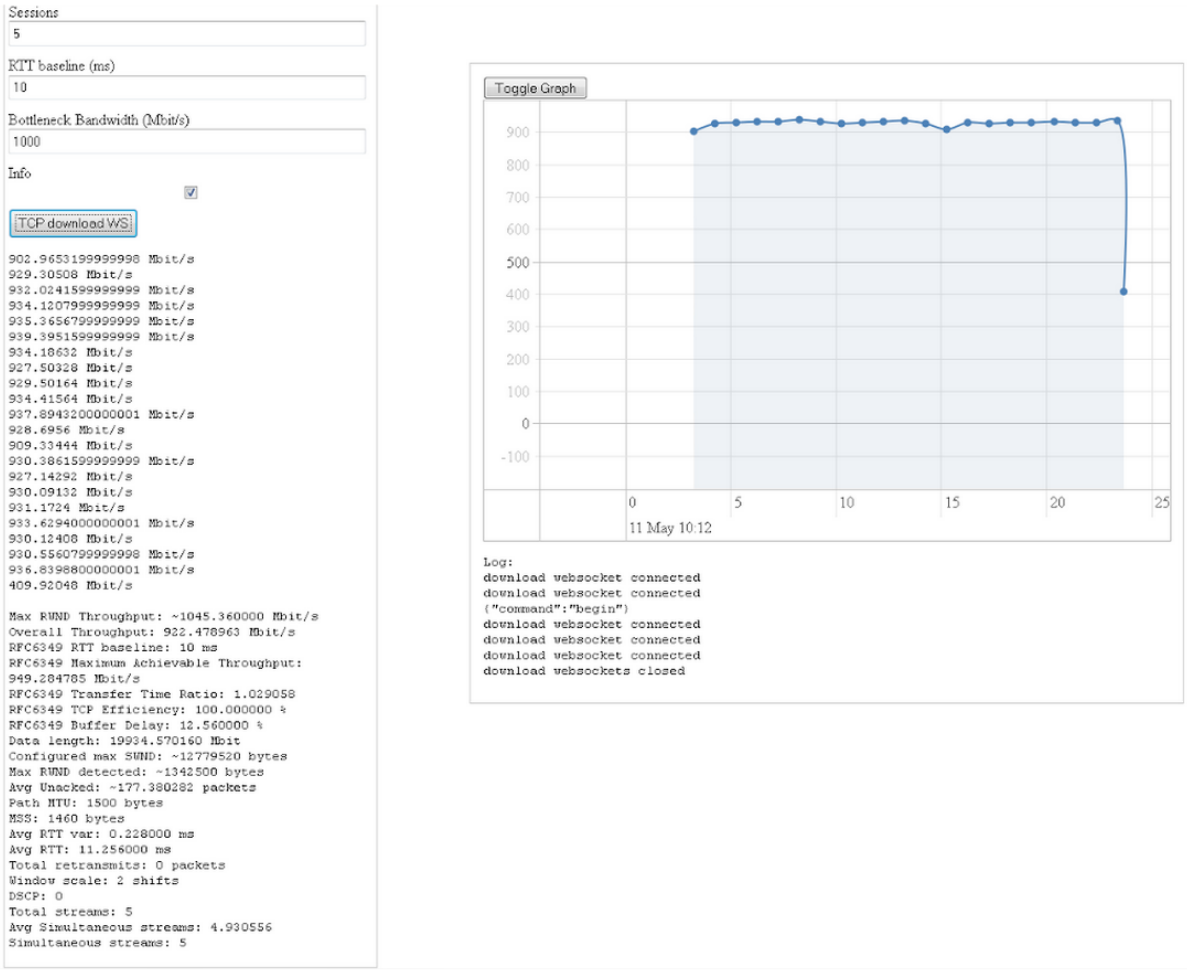


Figure B.1: Screenshot of the Throughput Download Test with Websockets.

Appendix C

WebRTC Proof of Concept Output Example

This was a rapidly developed proof of concept for performing the initial evaluation of the WebRTC data channel technology in regards of network performance. The metrics were measured over the application layer. See figure C.1 for a screenshot.

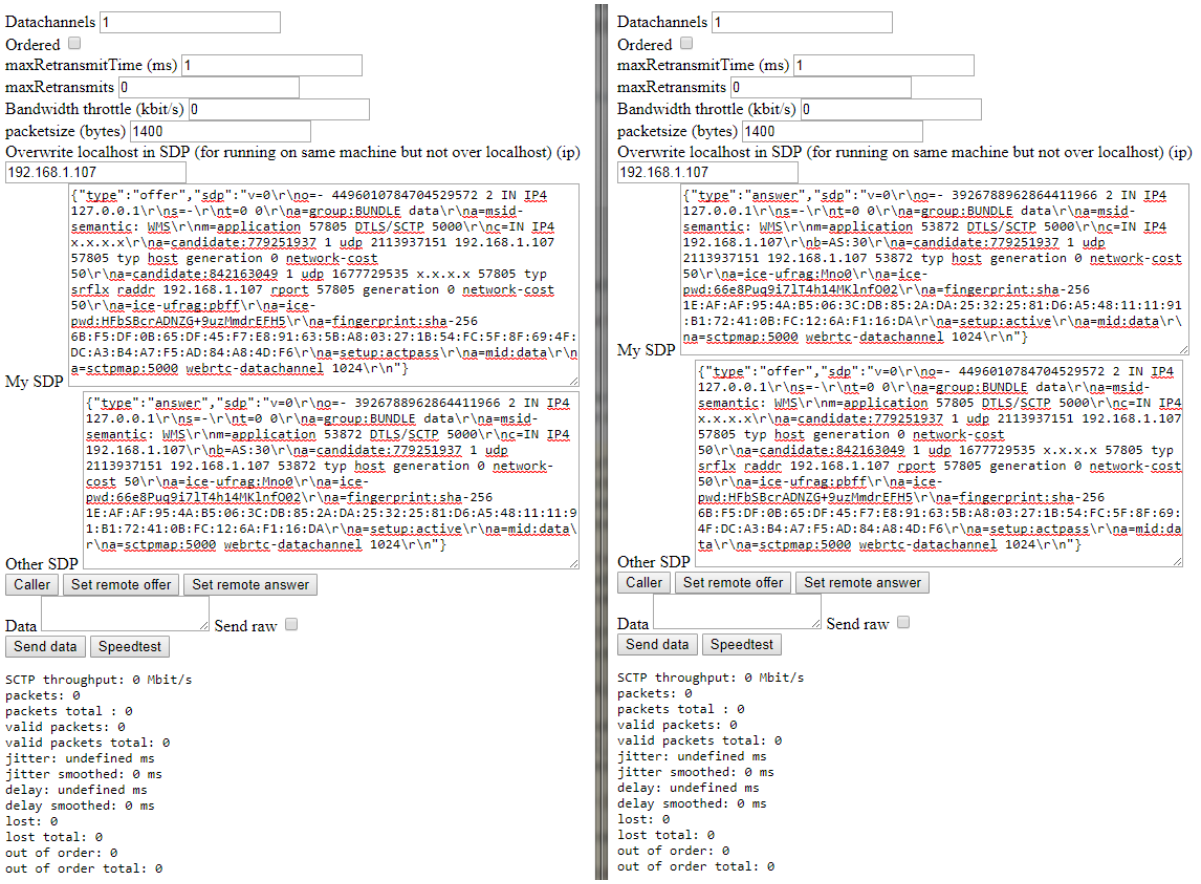


Figure C.1: Screenshot of the browser to browser proof of concept application when a connection have been established between two browsers on the same computer.

References

[1] OOKLA, "Speedtest.net by ookla - the global broadband speed test." http://www.speedtest.net/, 2015. (Accessed 2019-03-19).

[2] .SE (The Internet Infrastructure Foundation), "Bredbandskollen." http://www.bredbandskollen.se, 2015. (Accessed 2019-03-19).

[3] Adobe, "Flash player socket master policy files." https://www.adobe.com/devnet/flashplayer/articles/fplayer9_security.html #_Configuring_Socket_Policy. (Accessed 2019-03-19).

[4] M-Lab, "Ndt (network diagnostic test)." http://www.measurementlab.net/tools/ndt, 2015. (Accessed 2019-03-19).

[5] Speed Guide Inc., "Sg tcp/ip analyzer." http://www.speedguide.net/analyzer.php, 2015. (Accessed 2019-03-19).

[6] T. Hain, "Architectural implications of nat." RFC 2993, 2000.

[7] Information Sciences Institute University of Southern California, "Transmission control protocol." RFC 793, 1981.

[8] S. D. J. Mogul, "Path mtu discovery." RFC 1191, 1990.

[9] R. G. R. S. B. Constantine, G. Forget, "Framework for tcp throughput testing." RFC 6349, 2011.

[10] J. Postel, "User datagram protocol." RFC 768, 1980.

[11] R. Stewart, "Stream control transmission protocol." RFC 4960, 2007.

[12] R. F. V. J. H. Schulzrinne, S. Casner, "Rtp: A transport protocol for real-time applications." RFC 3550, 2003.

[13] F. B. D. B. K. Nichols, S. Blake, "Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers." RFC 2474, 1998.

[14] F. B. J. Babiarz, K. Chan, "Configuration guidelines for diffserv service classes." RFC 4594, 2006.

[15] Fasterdata (2015 May), "Host tuning ms windows." https://fasterdata.es.net/host-tuning/ms-windows/, 2015. (Accessed 2019-03-19).

[16] R. R. R. Koodli, "One-way loss pattern sample metrics." RFC 3357, 2002.

[17] A. C. T. Friedman, R. Caceres, "Rtp control protocol extended reports (rtcp xr)." RFC 3611, 2003.

[18] P. C. C. Demichelis, "Ip packet delay variation metric for ip performance metrics (ippm)." RFC 3393, 2002.

[19] WHATWG, "Xmlhttprequest." https://xhr.spec.whatwg.org/, 2015. (Accessed 2019-03-19).

[20] M. F. M. L. B.-L. Fielding, Gettys, "Hypertext Transfer Protocol." RFC 2616, 1999.

[21] MozillaZine, "Network.http.max-connections." http://kb.mozillazine.org/Network.http.max-connections, 2015. (Accessed 2019-03-19).

[22] Browserscope, "Browserscope network." http://www.browserscope.org/?category=network, 2015. (Accessed 2019-03-19).

[23] A. M. I. Fette, "The websocket protocol." RFC 6455, 2011.

[24] WC3, "The websocket api." http://www.w3.org/TR/2011/WD-websockets-20110419/, April 2015. (Accessed 2019-03-19).

[25] Mozilla Developer Network, "Websockets." https://developer.mozilla.org/en/docs/WebSockets, May 2015. (Accessed 2019-03-19).

[26] C. W. Paul Adenot, "Web audio api." http://webaudio.github.io/web-audio-api/, April 2015. (Accessed 2019-03-19).

[27] WC3k, "Webrtc 1.0: Real-time communication between browsers." http://www.w3.org/TR/webrtc/, May 2015. (Accessed 2019-03-19).

[28] M. T. R. Jesup, S. Loreto, "Webrtc data channels." I-D (internet draft), 2015.

[29] J. C. M. S. V. Paxson, M. Allman, "Computing tcp's retransmission timer." RFC 6298, 2011.

[30] Linux Fundation, "netem." http://www.linuxfoundation.org/collaborate/workgroups/networking/netem. (Accessed 2019-03-19).

[31] Google Inc., "Chrome." https://www.google.com/chrome/. (Accessed 2019-03-19).

[32] Google Inc., “Webrtc native development.” http://www.webrtc.org/native-code/development, May 2015. (Accessed 2019-03-19).

[33] Stefan Ãlund, “OpenWebRTC Github DataChannel issue.” https://github.com/EricssonResearch/openwebrtc/issues/3. (Accessed 2019-03-19).

[34] B. Katzarsky, “katzarsky/websocket.” https://github.com/katzarsky/WebSocket, May 2015. (Accessed 2019-03-19).

[35] J. F. J. D. F. Q. K. G. J. E. Mark Gates, Ajay Tirumala, “Iperf.” https://iperf.fr. (Accessed 2019-03-19).

[36] Chen Tao, “clumsy.” https://github.com/jagt/clumsy. (Accessed 2019-03-19).

[37] Mozilla Foundation, “Firefox 37.0.1 Crash Report [@ sctp_handle_sack].” https://crash-stats.mozilla.com/report/index/62df408d-ae7b-4c5f-8047-4d23c2150506. (Accessed 2015-05-06).