Performance Evaluation of WebRTC Data Channels

Rasmus Eskola
School of Science
Aalto University
Espoo, Finland
Email: rasmus.eskola@aalto.fi

Jukka K. Nurminen
School of Science
Aalto University
Espoo, Finland
Email: jukka.k.nurminen@aalto.fi

***Abstract*—This paper covers a study on WebRTC data channel performance in current web browser implementations. The goal is to find out whether WebRTC data channels are usable today in web applications demanding throughput performance for data transfers consisting of arbitrary data. Performance is measured using a purpose-built web application and various simulated network conditions. Packet capture is used for further analysis. The results reveal that current WebRTC data channel implementations do not adjust the SCTP window size from the default setting. This results in bad performance when network conditions and especially latency is not close to perfect. Changing the window size results in significantly better performance on high latency links, but the observed throughput performance is still not ideal. We can conclude from the test results that current WebRTC data channel implementations are not yet ready for high performance requirements nor mobile environments where battery life is important. The browsers need their WebRTC data channel implementations optimized in order for the technology to become truly useful.**

Index Terms - WebRTC, WebRTC data channel, SCTP, web browser, web application, performance measurement

I. Introduction

This paper presents and discusses performance aspects in practice of WebRTC data channels. WebRTC is an emerging web technology that enables realtime, true peer-to-peer communications in modern web browsers without the need for any browser extensions. WebRTC can be split into two separate parts, data channels and media channels. The media channels provide media functionality, while the WebRTC data channel API allows web application developers to exchange arbitrary data in a peer-to-peer fashion, with a rendezvous server needed only during connection setup (this is called signaling). Peer-to-peer connectivity is in many cases possible to achieve even with NAT (Network Address Translation) units and firewalls between the peers.

WebRTC data channels have only recently been implemented experimentally in web browsers. Moreover, data channels in the Chromium web browser have until recently used another protocol entirely than what has been standardized [1]. As such, there is currently a lack of study about standards-compliant WebRTC data channels. [2]. WebRTC has been largely targeted towards media, and the browsers have already had implementations for WebRTC video and audio streams for a few years. Most WebRTC related research so far has focused on these video and audio streams, which are transported over another protocol entirely than what the data channels use (SCTP) [3]. Due to this fact, the research done on media streams is mostly irrelevant for studying the behaviour of data channels.

The goal of this paper is to find out if WebRTC data channels are usable today in web applications demanding high throughput performance, and to initiate a discussion about any discovered performance issues with WebRTC data channels in current experimental implementations. In this paper, the data channels have been tested with the latest development versions of both the Firefox and Chromium browsers at the time of writing. Focus is put especially on the Firefox browser. Performance through relay servers using TURN (Traversal Using Relays around NAT) will not be covered, only true peer-to-peer connections were tested.

The method used to benchmark WebRTC data channels consists of a simple web application that sends data. This application measures various performance aspects of the WebRTC data channel, and provides statistics as a result. The application was tested with various levels of simulated network latency. Network traffic generated by the application at various latencies was then dumped and analyzed both by packet capture tools and by extracting the raw SCTP (Stream Control Transmission Protocol) stream via Firefox debugging options, giving us an insight into what happens at the lower levels of the protocol stack.

The results reveal that current WebRTC data channel implementations never change the SCTP send and receive window sizes from their initial values, so a low default value is used in both tested browsers. As a direct consequence of this, data throughput rates take a significant hit even with relatively small network latencies, which can be problematic in applications that need to transfer large amounts of data. After increasing the window sizes, performance does improve in settings with latencies typical for the public Internet, but the performance is still not as good as when performing the same tests in local networks.

The paper first presents related work (sec. II), and general aspects of WebRTC connections. Then methods used to do measurements will be presented in detail (sec. III); setting up network latency emulation, presenting the web application used for benchmarking and how results were gathered. Finally (sec. IV), the test results are presented, discussed and analyzed to find out what work there is left to do (sec. V).

II. Related work

WebRTC is a very exciting technology because there has previously not been any viable way of doing true peer-to-peer communication in the web browser without third party browser extensions. As such there is naturally a lot of research and papers discussing WebRTC and the multitude of possibilities it brings. WebRTC standardization has been going on for a few years already, with the first published W3C drafts from 23 August 2011 [4].

Research and testing similar to the work presented in this paper has been performed on the WebRTC media streams, which use the RTP protocol and Receiver-side Real-time Congestion Control. Lozano [5] concludes that the media streams are able to maintain high throughput even at relatively high latencies for real time media (up to 200 ms). The media streams work well even with some minor packet loss.

Fund et al. [6] have researched the impact on WebRTC media streams in different wireless environments. They find that despite adaptation mechanics that are in place in WebRTC media streams, the tested implementation was not able to adapt quickly enough to changes in network conditions resulting in bad service quality. However, they found that even at fairly high average RTT (Round-Trip Time), the video bitrate can stay somewhat constant. This is interesting for our experiment, because the WebRTC data channel does display a drop in throughput at high RTT as opposed to the media channel experiments.

While these research results on media channels seem to point to a very well functioning protocol, they do not draw any conclusions for SCTP-based data channels, which from initial testing seem to work well at lower latencies but have trouble keeping up throughput rates when latencies increase. More experimentation and research has to be done in this area, because data channels are a useful part of WebRTC with the arbitrary data transfers they provide that media channels do not.

III. Measurements architecture

A. Network setup

All measurements are performed in a LAN, thus eliminating most interference in latency and packet loss that could happen over the public Internet. A high-end laptop and desktop PC is used, and they are connected to a consumer-grade gigabit switch via ethernet cables.

Both computers run Linux 3.15 and the latest available versions of the Chromium (38.0.2067.0) and Firefox (33.0a1) browsers at the time of writing. Note that any measurements will be performed with Firefox only, but Chromium was also tested in some cases.

In order to simulate network latency and packet loss a network emulator in the Linux kernel called ”netem” is used. Netem is controlled by a command line tool called ”tc”. Adjustable and repeatable artificial network latencies can be achieved with netem and tc.

B. Measurement methods

All generated network traffic was inspected with Wireshark. Traffic was gathered by the means of packet capture either directly from the network interface or by instructing Firefox to dump the SCTP stream before encrypting it. Merely capturing network packets is often insufficient when working with WebRTC, due to this encryption. By running Firefox with the following environment variables set we can turn on logging for the browser's SCTP and WebRTC data channel code:

```
NSPR_LOG_MODULES="SCTP:5,DataChannel:5"
NSPR_LOG_FILE=dclog.log
```

The logs are written to the file dclog.log. This log file will contain any debug prints from the relevant parts of Firefox source code, and also any sent/received SCTP packets. We can filter the SCTP packets from the log file and convert them into a file format that is readable by Wireshark with the following command:

```
grep SCTP_PACKET dclog.log\
    | text2pcap -n -l 248 -D -t\
    '%H:%M:%S.' - dclog.pcapng
```

Now we can open the ”dclog.pcapng” file in Wireshark and inspect the SCTP stream in its unencrypted form.

C. WebRTC benchmarking application

The throughput benchmark application is a very simple JavaScript application that performs the minimum amount of required steps to set up a data channel. The data channel API was used directly without any middleware libraries, and signaling is performed via WebSockets.

Before running the test, both peers have to launch the web application which connects them to the signaling server. Either peer can then initiate the data channel setup process manually.

After the signaling process and once a data channel has been set up, the initiating peer starts repeatedly sending a JavaScript Uint8Array filled with successive integers. An array size of 32 kilobytes resulted in the best performance in the test environment. The data channel API will buffer the array and send it in chunks. Sending larger arrays at once possibly leads to less overhead both in networking and processing, as can be seen if the array size is decreased dramatically; then the throughput will also decrease.

The WebRTC data channel API abstracts the data channel behind a JavaScript object representing the data channel. The data channel object contains an important property for throughput intensive tasks, bufferedAmount, which represents the number of bytes that have been queued by the browser for sending over the data channel [7]. For maximum data throughput we need to keep track of this value, because overfilling the buffer causes an error and allowing the buffer to empty means the data channel will stay idle at times and in that case we are not utilizing the full throughput capabilities of the data channel.

In order to keep the send buffer filled, the application repeatedly calls the data channel send method until the channel

bufferedAmount value has reached past a limit. In this test a value of 1024 kilobytes was found to work well on Firefox, with higher values possibly overfilling the send buffer and lower values causing the buffer to occasionally empty. This buffer filling loop function is called every time the JavaScript interpreter runs its event loop by putting it inside of a setTimeout call with zero timeout. By doing this instead of sending data in an infinite loop, we avoid making the browser unresponsive and possibly even freezing it depending on how the browser would handle an infinite loop in JavaScript.

The receiving peer will display average data throughput in MB/s. Data throughput is calculated by summing up the amount of received bytes and dividing by the time in seconds since the test was initiated. This value represents the effective average throughput, that is, how much useful data can be sent over the data channel.

IV. Performance evaluation

A. Data channel throughput performance results

High throughput over data channels seems to be very heavy on CPU usage, and the CPU load is largely single threaded. On a recent high-end PC, throughput readings of up to 30.0 MB/s could be observed in local testing from one browser process to another. At this point both browsers each put full load on their respective CPU cores. So at very high throughput we are CPU bound currently, even on a fast modern PC.

In the process of testing the data channel throughput performance, it was discovered that at very low RTT values the data channel is able to fully utilize the capacity of a 100 Mbit/s link as long as both peers have enough CPU power. In our test environment with one powerful desktop PC and one fairly powerful laptop PC connected with a gigabit link, a 18.1 MB/s throughput could be maintained with one of the laptop's CPU cores on full load. Interestingly, it seems that receiving is more CPU intensive than sending, as when the laptop was on the receiving side a throughput of only 15.5 MB/s was observed.

With the laptop on the sending side, desktop PC on the receiving side, the following Wireshark IO graph was obtained via the means of packet capture, as seen in figure 1.

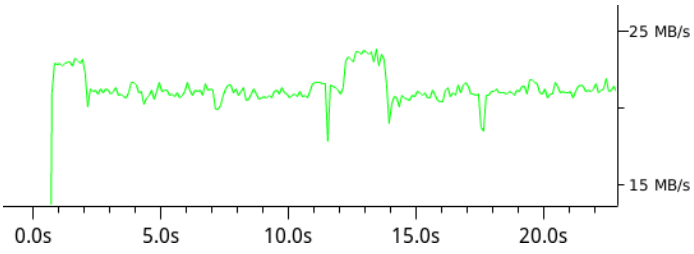

Fig. 1. Throughput as measured with Wireshark in low-latency conditions

From figure 1 we can obtain the total throughput which is slightly above 20.000.000 bytes/s or roughly 19 MB/s. The average effective throughput as reported by the benchmarking application was 18.12 MB/s. The line represents data flowing from the sender to the receiver. There is also a small flow of packets from the receiver to the sender not shown in the figures. It consists mostly of SCTP SACK (Selective Acknowledgement) messages as will be seen later when the unencrypted SCTP stream is analyzed.

B. Data channel throughput with simulated latency

If we perform the exact same test but with 50 ms of simulated RTT by netem, we will get very different results, as can be seen from figure 2.

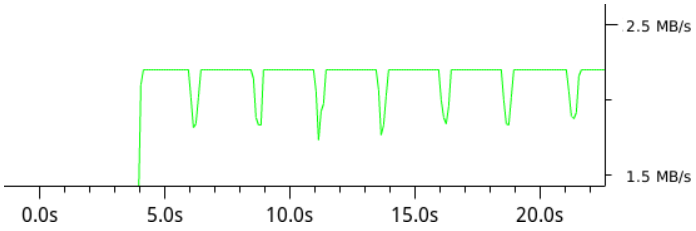

Fig. 2. Throughput as measured with Wireshark, 50 ms simulated RTT

Data throughput drops by an order of magnitude from 18 MB/s to under 2 MB/s. The average effective throughput as reported by the benchmarking application was 1.82 MB/s.

If we zoom in on the graph (tick interval 0.001 s) we can see a repeating pattern of short data bursts every 50 ms, see figure 3.

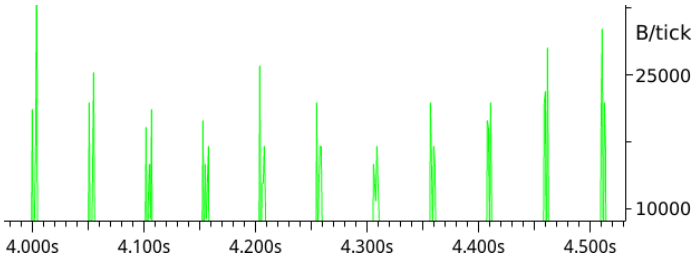

Fig. 3. Throughput as measured with Wireshark, 50 ms simulated RTT, zoomed in

The captures have been performed on the receiver's side. The results show that the sender has to wait for acknowledgement messages from the receiver before sending more data.

If we dump the unencrypted SCTP stream as explained previously, instead of only performing a packet capture on the DTLS-encrypted SCTP stream, we notice something interesting. Neither the Chromium nor Firefox current implementations of the data channel API adjust the SCTP send and receive window sizes. Instead the SCTP userspace implementation's default value of 128 KiB window size is used, as is evident from the SCTP stream dump from a Firefox peer; a_rwnd which is the advertised receive window size never changes from the default of 131072 bytes. This relatively low window size causes a great drop in throughput performance when there is network latency between the peers. Further increasing the latency causes additional reductions in data throughput, even though the network link capacity stays constant.

The reason behind this is that the small send and receive windows quickly get filled with data. At this point in SCTP, just like in TCP, the sender must wait for an acknowledgement

message from the receiver before more data can be sent. Thus the network link stays under-utilized for long periods of time.

According to the bandwidth-delay-product [8], the maximum amount of unacknowledged bytes in flight is equal to:

$$\text{BDP} = \text{Max Throughput} \cdot \text{RTT} \tag{1}$$

If the entire window can be acknowledged in one RTT, the maximum value for data throughput is:

$$\text{Max Throughput} = \frac{\text{Window size (BDP)}}{\text{RTT}} \tag{2}$$

As we can see from equation 2, at constant window sizes the maximum throughput is inversely proportional to the RTT. Doubling the RTT value will halve the maximum throughput. At the default window size of 128 KiB, table I contains the observed throughput results for different simulated RTT values.

RTT	Effective Throughput	Total throughput	Theoretical max
10 ms	8.84 MB/s	9.88 MB/s	12.5 MB/s
20 ms	4.67 MB/s	5.22 MB/s	6.25 MB/s
40 ms	2.37 MB/s	2.65 MB/s	3.13 MB/s
80 ms	1.20 MB/s	1.35 MB/s	1.56 MB/s
160 ms	0.60 MB/s	0.67 MB/s	0.78 MB/s
320 ms	0.30 MB/s	0.34 MB/s	0.39 MB/s

TABLE I
THROUGHPUT AT DIFFERENT RTT WITH DEFAULT WINDOW SIZE

We can see that doubling the simulated RTT indeed does halve the observed throughput. We also can see that the measured throughput rates stay fairly close to the theoretical maximum rates. The table shows that throughput performance gets unacceptably low with anything but the lowest latency links for any kind of bulk data transfer. Worse yet, the observed throughput performance at RTT values near those of cellular technologies such as 3G and 4G is much lower than the bandwidth available using these technologies in our testing. This has reprecussions in energy consumption on mobile devices because for bulk data transfer the radio has to stay on for longer than would otherwise be needed.

V. THROUGHPUT WITH A LARGER SCTP WINDOW

By modifying the source code of the Firefox browser we changed the window size of the SCTP sockets when data channels are initialized. The default window size value of 128 KiB was raised to one megabyte, allowing for a much larger RTT before any throughput degradation should be possible to observe in a 100 Mbit/s network, which is a good target considering the CPU limitations discussed in the previous section.

The same throughput test at 50 ms was performed once more now with the modified Firefox browser. The resulting throughput graph can be seen in figure 4.

We can tell from the throughput graph that performance is considerably better than before with a simulated RTT of 50 ms. We are now able to reach average throughput speeds of approximately 10 MB/s at 50 ms RTT.

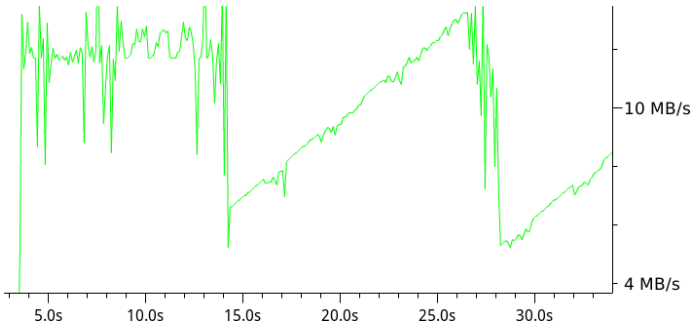

Fig. 4. Throughput as measured with Wireshark, 50 ms simulated RTT, with modified version of Firefox

At higher RTT, we can also see that the SCTP association struggles to keep up the high throughput speeds in its congestion avoidance phase like it does at smaller RTT. The sawtooth-like wave indicates that SCTP is in congestion avoidance. While the congestion control algorithm tries to increase the data transmit rate, at some point congestion is detected and the transmit rate is halved, resulting in the sawtooth-like wave that can be seen. This sawtooth pattern continues for the duration of the data transfer.

Looking at the dumped SCTP stream we can see that right before the data transmit rate is halved, the value of cumulative_tsn_sack in SCTP SACK packets will stop incrementing. The cumulative TSN (Transmission Sequence Number) ACK marks the last consecutive TSN of received DATA packets on the receiver side. As per RFC 2960 [9] this value will acknowledge all smaller or equal TSNs. This behaviour indicates a dropped SCTP DATA packet. A retransmitted packet with the correct TSN can be seen roughly one RTT later, which indicates that a fast retransmit has occurred. Still the SCTP stream will halve its throughput rate.

When performing a SCTP stream dump on both the receiving and sending sides, we can see that packet loss indeed does occur. On the sender side the dump shows that a packet was sent, but on the receiver side it was never received. Oddly enough packets were dropped in the experiment, even though the throughput rates were significantly under the maximum rates of the network. The network was also tested with normal TCP and UDP connections, which were able to perform at up to half the maximum throughput rate of the test network without any signs of packet loss especially in the UDP case. TCP connections were able to gracefully handle the minor packet loss at higher throughputs, and reach througphut rates of up to 1 Gbit/s, which hits the hardware limitations of the test network.

SCTP was also tested outside the browser implementations in a program called iperf. Iperf does not normally support SCTP, but there are patched versions available that use the operating system's (Linux in this case) own kernel SCTP implementations. Similar results were found here, throughput performance is lowered a lot when the latency between hosts increases. This is possibly a wider issue with SCTP than only the browser WebRTC datachannel implementations.

Increasing the default SCTP window sizes for WebRTC data channels is a must. As we can see from the results, problems aside with the congestion control mechanism, performance still did improve by a huge margin. There are however potentially some issues with large window sizes, such as a big window potentially having to be resent if packet loss does occur. This may lead to longer delays in delivering already received data to the application, since we are stuck waiting for the lost packet or worse yet entire window to be retransmitted. A better compromise between throughput and potential latencies when packet loss does occur needs to be found. The WebRTC developers clearly preferred low latency over throughput when selecting the default value or didn't want to commit to any default values yet.

VI. Future work

WebRTC data channels have potential to become a very useful technology in enabling cross platform, secure, high performance WebRTC applications, but there are still a few obstacles to overcome. Research has to be performed not only in measuring throughput rates, but also requirements for processing power, battery life on mobile devices and possibilities of optimizing the SCTP protocol and also the implementations of the WebRTC datachannel protocol in areas that have not yet been tested very much such as wireless environments.

A lot of experiments that have been performed on the WebRTC media channels still are necessary to also perform on WebRTC datachannels due to the protocol differences. Optimizing WebRTC to be latency and packet-loss tolerant needs to be of high priority for WebRTC developers in the near future. Performance in different network conditions such as cellular and wireless networks with varying packet loss and latencies and measurements in other similar realistic scenarios has to be analyzed more closely to draw conclusions on real-world WebRTC application performance.

It would be desireable to implement an adaptive window scaling mechanism into the browsers' implementations of SCTP data channels or at least increase the default window size as discussed. WebRTC is already experimented with by some early adopters, but for its success it is essential that WebRTC works just as fast as we would expect a TCP connection to perform. There are currently possibilities to dramatically increase the data channel performance for a lot of use cases, especially in the mobile world where currently, WebRTC application performance and battery life suffers.

VII. Acknowledgements

This work was partly supported by Tekes (The Finnish Funding Agency for Technology and Innovation) project Everyday Sensing.

References

[1] Google Inc., "WebRTC Status - Chrome," 2014, webrtc.org WebRTC status. Cited 11.8.2014. [Online]. Available: http://www.webrtc.org/chrome

[2] M. R. Jesup, E. S. Loreto, and M. U. o. A. S. M. Tuexen, "WebRTC Data Channels," IETF, Internet-Draft, Jun. 2014, work in progress. [Online]. Available: http://tools.ietf.org/id/draft-ietf-rtcweb-data-channel-11.txt

[3] C. Jennings, T. Hardie, and M. Westerlund, "Real-time communications for the web," *Communications Magazine, IEEE*, vol. 51, no. 4, pp. 20–26, 2013.

[4] A. Bergkvist, D. C. Burnett, C. Jennings, and A. Narayanan, "WebRTC 1.0: Real-time Communication Between Browsers," W3C, Editor's Draft, obsoleted, Aug. 2011, obsoleted. [Online]. Available: http://dev.w3.org/2011/webrtc/editor/archives/20140617/webrtc.html

[5] A. A. Lozano, "Performance analysis of topologies for web-based real-time communication (webrtc)," Master's Thesis, Aalto-university, School of Electrical Engineering, Espoo, 2013, available: https://aaltodoc.aalto.fi/bitstream/handle/123456789/11093/master_Abell%C3%B3_Lozano_Albert_2013.pdf. Cited 13.8.2014.

[6] F. Fund, C. Wang, Y. Liu, T. Korakis, M. Zink, and S. S. Panwar, "Performance of dash and webrtc video services for mobile users," in *Proceedings of the 2013 20th International Packet Video Workshop, San Jose, California, USA*, 2013, pp. 1–8.

[7] A. Bergkvist, D. C. Burnett, C. Jennings, and A. Narayanan, "WebRTC 1.0: Real-time Communication Between Browsers," W3C, Editor's Draft, Jun. 2014, work in progress. [Online]. Available: http://dev.w3.org/2011/webrtc/editor/webrtc-20110823.html

[8] K. Chen, Y. Xue, S. H. Shah, and K. Nahrstedt, "Understanding bandwidth-delay product in mobile ad hoc networks," *Computer Communications*, vol. 27, no. 10, pp. 923–934, 2004.

[9] S. et al., "RFC 2960: Stream Control Transmission Protocol," Oct. 2000, status: Proposed standard. [Online]. Available: http://tools.ietf.org/html/rfc2960