CECS

CENTER FOR EMBEDDED & CYBER-PHYSICAL SYSTEMS

WebRTCBench: A Benchmark for Performance Assessment of WebRTC Implementations

Sajjad Taheri, Laleh Aghababaie Beni, Alexander V. Veidenbaum, Alexandru Nicolau,

Rosario Cammarota, Jianlin Qiu, Qiang Lu, Mohammad R. Haghighat

Center for Embedded and Cyber-Physical Systems

University of California, Irvine

Irvine, CA 92697-2620, USA

{sajjadt,laghabab,alexv,nicolau}@uci.edu, rosarioc@qti.qualcomm.com,

{jianlin.qiu,qiang.lu,mohammad.r.haghighat}@intel.com

CECS Technical Report CECS-15-01

April 20, 2015

WebRTCBench: A Benchmark for Performance Assessment of WebRTC Implementations

Sajjad Taheri, Laleh Aghababaie Beni, Alexander V. Veidenbaum, Alexandru Nicolau
Dept. of Computer Science, UC Irvine, Irvine, CA
Email: {sajjadt,laghabab,alexv,nicolau}@uci.edu

Rosario Cammarota
Qualcomm Research
Email: rosarioc@qti.qualcomm.com

Jianlin Qiu, Qiang Lu, Mohammad R. Haghighat
Intel Corporation
Email:{jianlin.qiu,qiang.lu,mohammad.r.haghighat}@intel.com

***Abstract*—WebRTC is an HTML5 API that allows browsers to establish a peer-to-peer connection for transferring data and media content via JavaScript APIs. This functionality enables broad range of new applications to emerge and is going to revolutionize Web communication. However, this technology is still under development and standardization process. Hence, detecting performance bottlenecks of different implementations across operating systems and architectures can help improve it significantly, and a benchmark suite would be a great help to accomplish this task. In this paper, we present WebRTCBench, a benchmark which measures WebRTC performance for establishing peer to peer media streams and data channels and also WebRTC media and data communication. We present and discuss performance evaluation of WebRTC implementations across a range of architectures and operating systems. This benchmark is publicly available under GPL license.**

Keywords: WebRTC, Peer-to-peer Connection, Performance Measurement

I. Introduction

WebRTC is result of a joint effort between the World Wide Web Consortium (W3C) and Internet Engineering Task Force (IETF) to provide real-time communication capabilities in browsers. WebRTC aims to fill a critical gap in web application design by providing APIs to access native devices, share streams of audio/video and establishing bi-directional data channels. It consists of multiple underlying communication protocols and standardized JavaScript APIs which expose unique browser capabilities to web developers [8]. It's also an industry effort and major browser vendors such as Mozilla and Google provide WebRTC support in their browsers.

WebRTC API is divided into three main categories:
(1) **GetUserMedia**: provides access to user media including cameras, microphones and display.
(2) **DataChannel**: allows arbitrary data to be transferred through a peer to peer data channels.
(3) **RTCPeerConnection**: sets up a direct connection between two WebRTC enabled end points, allowing media streams and data channels to be attached to it.

TABLE I: Sample WebRTC applications, considering GetUserMedia (GM), RTCPeerConnection (PC), DataChannel(DC)

Application	GM	PC	DC
Media Recording, Gesture Recognition, Voice Translation	✗		
Voice Call, Video Conferencing, Screen Sharing	✗	✗	
Messaging, File Sharing, Data Transfer		✗	✗
Streaming Local Media files		✗	

WebRTC novel capabilities enable browsers to have a wide range of new applications using web standards. Multiple web applications have emerged allowing users to host conversations and share videos and data. Online multi-player games such as Banabread [1] are now benefiting from peer-to-peer data transfer provided by data channels. Table I lists some applications of WebRTC along with the components of WebRTC that are used by them. Each class of applications requires certain classes of API functions. An extension to WebRTC media capture has been proposed which provides the ability to acquire and transfer depth streams synchronized with color video from Kinect-like cameras[2]. RGB-D signals have boosted research in computer vision and allow efficient and more precise applications such as skeleton tracking and 3d scanning. We believe that with access to depth camera, perceptual interfaces for web applications will emerge adding another level of improvement to user experiences.

Direct interaction of WebRTC with users make its response time and performance critical. WebRTC's high computational and bandwidth requirements, especially on mobile devices with limited resources, calls for improvement. Measurement of basic and advanced functionalities of current and future implementations of WebRTC is particularly important at this stage of standardizatoin and development.

Different aspects of a WebRTC systems can be evaluated. For instance, Singh, et al., showed how WebRTC congestion control performs under different topologies and latencies[11]. Another study demonstrated how WebRTC video quality changes in a realistic setup, as devices are moving around the

network[6]. The main focus of aforementioned works is the network and its performance.

Main goal of designing WebRTCBench (available at https://github.com/INTCUCI/WebRTCBench) is to spot critical performance bottlenecks of WebRTC implementations which is an important step for guiding optimizations. This benchmark evaluates performance of WebRTC functionalities considering different implementations, architecture and platforms as each of these factors can have big impact on performance. The other goal of WebRTCBench is to provide a cross-platform benchmark using only JavaScript API and without any browser modifications.

Using representative benchmarks for evaluating performance of computing devices is a well-known approach. Although there are several benchmarks for evaluating JavaScript execution performance, such as [3], WebRTCBench is the first benchmark that evaluates WebRTC functionalities and allows quantitative comparison between WebRTC implementations across browsers, devices and operating systems.

II. WebRTC Benchmark

This section describes how WebRTCBench is designed and implemented. The selection of performance primitives measured by the benchmark will be justified. We also describe how performance statistics are collected.

Figure 1 depicts a communication system based on WebRTC triangle. It consists of WebRTC enabled peers and a web/signaling server. Web server is hosting the WebRTC application while signaling server is used by peers to negotiate on content and settings, and coordinate the connection. In case of a multi party WebRTC session, different topologies can be used. It is possible to establish a connection between each pair in a full-mesh scenario which offers the lowest latency but with the cost of high bandwidth usage. Alternatively, media servers can be placed in middle of connections, making a star-like topology or use routing technique. This will increase the latency and needs a more complicated logic in the middle layers but requires less bandwidth. A WebRTC system regardless of topology is a collection of peer-to-peer connections. In this paper we are interested in measuring the performance of WebRTC peer-to-peer connections.

WebRTCBench is composed of three components to make a WebRTC system similar to one depicted in Figure 1: a web/signaling server application, a WebRTC application(actual benchmark application) and a database hosted at the server.

A. Performance Measurements

The current version of benchmark measures performance of two complementary aspects of a data/media peer to peer communication provided by WebRTC: connection establishment and media engine/data channel performance during video calls and data transfers. With our benchmark, it is possible to exercise specific portion of WebRTC API. Since the depth-related API is not formally implemented yet, we haven't added them into the benchmark yet, but they will be considered for future implementations.

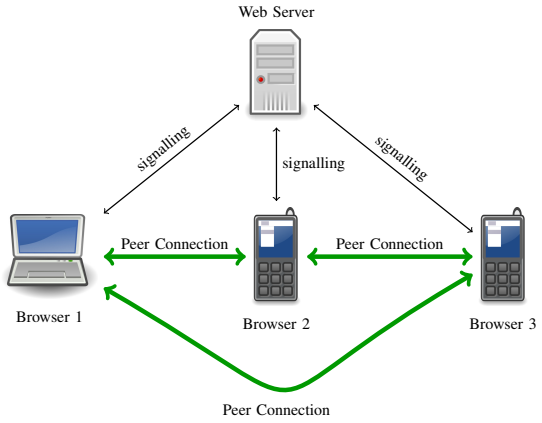


Fig. 1: A Typical WebRTC system

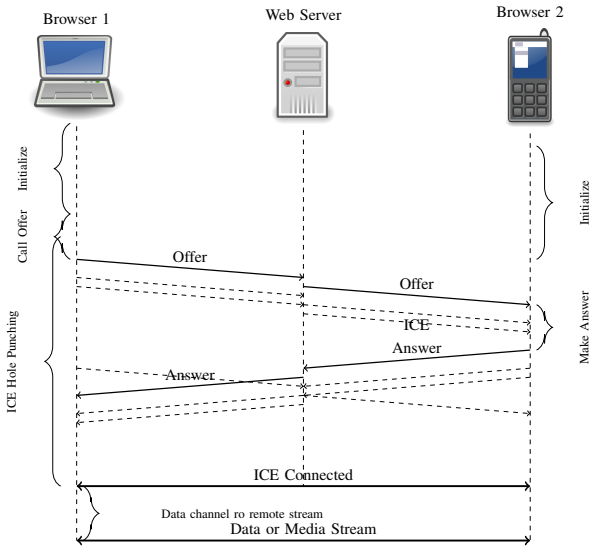


Fig. 2: WebRTC peer connection flow between two peers

1) Connection Establishment: This section describes necessary steps for establishing a WebRTC peer connection between two peers and highlights basic functionalities involved. WebRTC connection establishment flow between two peers is shown in figure 2. By identifying and minimizing time consuming steps, peer connection can be established faster.

For peer-to-peer connectivity, peers need to know each other's address. However, it's very likely that peers are behind NATs and network firewalls blocking access from outside. A technique called ICE hole punching [9] is used during which peers gather their public address candidates using STUN servers and exchange them through the signaling server. Different NAT types make specific constraints on reachability from outside [10]. Hole punching is effective method for connectivity for all cases of NAT types except for Symmetric NATs.

WebRTC doesn't mandate a specific messaging mechanism for signaling and a wide variety of options, including Websockets[5] and XmlHTTPRequest[14], can be used for this purpose.

As depicted in Figure 2, first, each peer instantiates an `RTCPeerConnection` object. Optionally, they can acquire local media streams which can be used by a media element for local display or be attached to `RTCPeerConnection` object for transmission to the other peer. Peers also need to establish a connection with the signaling server which they use to communicate messages. At this point, one of the peers (the caller) makes an offer indicating the content and settings of the communication and sends it to the other peer using the signaling server and starts the hole punching sequence. On the other side, when the callee receives the offer through the signaling channel, it prepares an answer and sends it back, and will start the ICE hole punching. ICE hole punching is a technique in which two peers exchange the IP addresses gathered from STUN and TURN servers and check it for connectivity constantly until they find a way to connect. We consider a variant of this technique in which both peers look for finding ways to connect incrementally [7], which is implemented in all browsers. After successful connection, a data channel can be opened or a remote media can be played. A portion of connection setup time is spent on the server, but our experiments show that the corresponding time is negligible. Depending on the required functionality of WebRTC, a subset of these parameters contribute to total time of establishing a WebRTC connection. WebRTCBench records the time different events occur and uses them to calculate the delay for each step. We have categorized and listed all the steps in Table II.

TABLE II: Connection flow measured primitives

	Parameter	Definition
Initialization	New Peer Connection	Time for instantiating a new `RTCPeerConnection` object.
	Get User Media	Time spent capturing user media excluding time waiting for user permission.
	Can Play Local Media	Time spent after media request until local media becomes available.
	Setup Signaling Channel	Time to setup signaling channel using WebSockets.
Communication	Make Call	Time for setting session descriptions and creating offer.
	ICE Hole Punching	Time spent for ICE hole punching process.
	Make Answer	Time for setting session descriptions and creating answer.
	Signaling	Time spent propagation of answer/offer messages and signals across the network.
	Open Data Channel	Time for opening data channel.
	Play Remote Stream	Time for play remote media stream to be available.

2) WebRTC Media Engine: Figure 3 shows the media engine block diagram used in current WebRTC implementations. After `RTCPeerConnetion` setup and call establishment, a complete media pipeline is in place with peers performing capturing/decoding/ or/and encoding/rendering at the same time.

WebRTC Stats API is leveraged to obtain performance parameters associated with different components of media engine including encoder, decoder, jitter buffer and network. Renderer performance on the other hand, is evaluated by recording the number of frames that are dropped/painted by HTML video element and their resolution. Table III lists parameters from each module that are recorded by WebRTCBench.

Benchmark input media can come from two sources: 1) input devices 2) fake devices. With WebRTCBench it's possible to get stream from a specific media device (for example front or rear camera of a smart phone) which satisfies user provided constraints. Constraints that are currently supported are video resolution and frame rate. Most browsers can generate fake audio/video streams for testing purposes. Additionally, uncompressed video files can be used as fake devices. Using fake devices however may require starting a browser with proper command line switches.

TABLE III: Media engine performance statistics

Module	Measured Statistics
Video Capture	Captured video resolution; And number of frames captured per second.
Video Encoding	Encoded stream bitrate, frame rate and resoloution;also encoder ramp-up speed
Video Decoding/ Rendering	Frame numbers decoded/rendered per second; resolution of rendered stream .
Networking	Bytes/packets sent, received and dropped, RTT.
Jitter Buffer	Jitter, A/V Sync.

Since most WebRTC implementations start up streaming with low encoding/sending bitrate and ramp up to meet the preset bandwidth constraint, WebRTCBench monitors the actual encoding bitrate, and starts recording the performance data when the encoding bitrate has been stabilized. Figure 12 shows typical ramp up curves for send bitrate.

The first WebRTC proposal mentioned VP8 as the only mandatory codec for WebRTC endpoints to implement. However due to industry support and release of OpenH264 implementation, H264 also become a mandatory codec to implement. At the time of writing this paper H264 WebRTC support is added to the latest Firefox release for both Desktop and Mobile platform. On the other hand, VPx family of codecs are under development with rapid pace and latest Chromium Canary is shipped with VP9 communication capabilities. WebRTCBench can select the video/audio codec used in the experiments among VP8, H264 and VP9 codecs if supported by implementation.

3) WebRTC Datachannel: WebRTCBench can be used to evaluate the performance of data channels. Two metrics measured by benchmark are transfer bandwidth and latency. Applications such as peer-to-peer multiplayer games require low latency whereas data transfer requires high throughput. Table IV lists performance stats collected by WebRTCBench. Measurements are done at two levels: Application level in which measurements are only with consideration of user messages and Network level in which statistics are collected based on actual network performance. Performance of different configurations of Data channels such as reliable or unreliable modes with out of order transmission is supported with help of data channel constraints.

TABLE IV: Datachannel measured statistics

Category	Measured Statistics
Network	Bytes sent/received/RTT
Application	Bessages sent/received /RTT

B. Benchmark Challenges

This section lists some of the challenges that we faced during benchmark development and workarounds that we used.

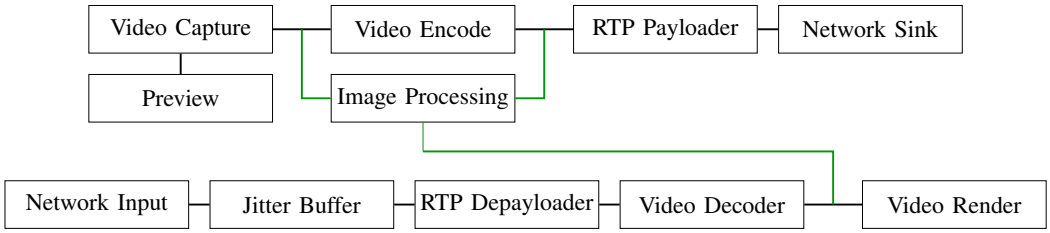


Fig. 3: WebRTC media engine

1. GetUserMedia returns the stream after receiving user permission. Hence the reported time will include the time spent waiting for user permission. We bypassed Google Chrome user permission by accessing the server through a HTTPS connection and then adding our server as a media exception for which browser can remember the permission. Mozilla Firefox on the other hand has a flag for disabling user permission dialog.

2. WebRTC specification is still in flux and changes over time. Many functions are prefixed and some functionalities have to be implemented differently on each implementation i.e. collecting statistics or file transfer.

3. Benchmarking is performed on two nodes with potentially unsynchronized clocks. Some measurements such as data transfer time should be done on both peers. We used data channels to synchronize the clock. The following procedure is repeated for a large enough number of iterations. At i_{th} step, one of the clients sends a ping message and records the current timestamp at $ts1_i$. The other client will send the pong message with its current timestamp tr_i as soon as it receives a ping message. Originating client will again record its current timestatmp as $ts2_i$ when it receives the pong message. The minimum value of $ts2_i - ts1_i$ over all iterations is considered to be the round trip time rtt. Consequently, the clock difference between two peers is $tr_j - ts1_j - rtt/2$ where j points to the iteration with the lowest round trip delay.

4. Our benchmark measures statistics about media engine performance. However, for a true comparison same video sequences should be used. Live videos from GetUserMedia have variations and it is impossible to have a unique input for each tests. Luckily, browsers support synthetic streams or raw uncompressed video files as fake devices that were used in our experiments.

III. Impelementation

WebRTCBench has three main components to make a typical WebRTC system: a unified web and signaling server, a benchmark application written in HTML5 and an optimal database.

A. Web/Signaling Server

We implemented a server which acts as both a web server and a signaling server using Node.js [12] platform. Once running, peers navigate to the server URL with their browsers to download the benchmark application and exchange necessary signaling messages to establish the connection. After completion of experiments, server collects results from peers and stores them in text format and in a database for further analysis.

B. WebRTC Benchmark Application

A fully functional WebRTC application using HTML5 components encompassing main functionalities of WebRTC, such as audio/video call, file sharing and messaging is implemented. It currently supports Google Chrome, Mozilla Firefox and Opera for Desktop and Android platforms which is almost 65% of installed browsers on the market[4]. Interoperability between all browsers is also provided.

To use the benchmark, users navigate to the benchmark URL address with their WebRTC enabled browser and download the WebRTC benchmark application. The application lets users decide on settings for their experiments. They can select any combination of text conversation, audio/video conferencing or file sharing along with the constraints for connections such as preferred video codec, captured video resolution, and data channel constraints. Users provide their own machine description including: device, processor and connection type. Performance evaluation is done at both sides of the conversation in parallel and users are informed through a provided console. WebRTCBench has an automated mechanism to collect experiments information from peers' browsers. This information is stored in database along with performance results for future analysis.

WebRTC Javascript APIs has made it possible to collect performance statistics without any modification to the existing browsers. In order to measure the performance of functionalities listed in Tables II, III and IV the source code is instrumented and multiple callback functions for capturing events are placed. Application at each peer uses these events to generate and report performance statistics to the server after establishment of connection. WebRTC media engine exposes a series of statistics through its JavaScript object attached to `RTCPeerConnection` instance. Our benchmark suite utilizes these statistics interface to gather media engine performance data.

C. Database

WebRTCBench provides necessary modules and scripts to build and access a MySQL database. Storing all performance information along with corresponding experiment configuration in a database allows further analyzing of WebRTC performance. It also makes it possible to have a comparison between different versions of WebRTC. The database is designed to be extensible in order to cope with future WebRTC features.

IV. Evaluation

This section presents performance results obtained using the WebRTCBench to demonstrate its capabilities. Experiments are conducted to provide insights for understanding of the WebRTC behavior and to demonstrate that WebRTCBench is a useful tool for developers and researchers. Experiments cover most of the capabilities provided by benchmark, i.e. peer to peer connection establishment and datachannel/video call performance. Experiments were conducted on two major WebRTC enabled browsers (Google Chrome, Mozilla Firefox) running on four different configurations shown in Table V. In order to have a rough estimate of JavaScript execution and network performance of our configurations, scores of Octane JavaScript benchmark suite [3] is collected. Higher Octane score means better average JavaScript performance. Each reported experiment was repeated ten times and the average performance are reported. To avoid any influence from previous runs, browsers are completely restarted with a clean profile before each experiment. Throughout the rest of paper, letters 'C' and 'F' in figures represents Google Chrome and Mozilla Firefox respectively. We also considered two different networks in our experiments listed in Table VI .

TABLE V: Specifications of test devices

Name	Processor	O.S.	Browser	Octane Score
Razr i	Atom, 2 GHz	Android 4.1.2	Chrome 41	3085
			Firefox 36	2852
Xperia Z3	Qualcomm 2.5 GHz, Quad Core	Android 4.4.4	Chrome 41	4436
			Firefox 36	4194
PC 1	X86 Core i7 2.4GHz	Windows 8.1	Chrome 41	22520
			Firefox 36	20010
PC 2	X86 Core i7 1GHz	Windows 8.1	Chrome 41	5131
			Firefox 36	4839

A. Initialization and Peer Connection Establishment Performance

Figure 4 shows initialization performance for different configurations of browsers and devices. This figure only represents the time that is needed for instantiation of a `RTCPeerConnection` object as setting up signaling, and creating a `RTCDataChannel`is negligible on all platforms (atmost a few milliseconds). This time depends on a number of factors but mainly the operating system and available resources. Generally, it is considerably faster on Google Chrome implementations compared to Mozilla Firefox.

Figure 5 shows the time spent for capturing user media stream consisting of both audio and video with a fixed resolution of 640×480 pixels. Captured stream can be attached

TABLE VI: Networks used in experiments

Network name	NAT Type
UCINet	Full Cone
TMobile US	Restrict

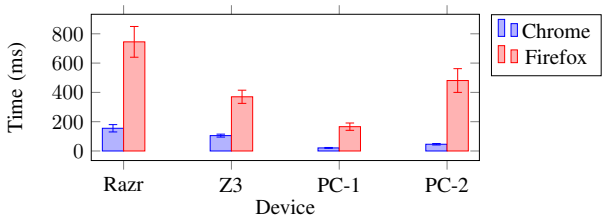


Fig. 4: Peer connection initialization performance

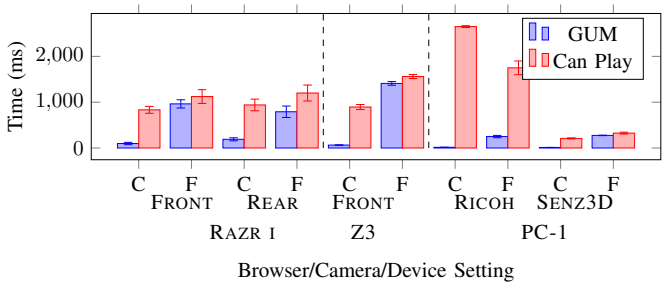


Fig. 5: Get user media delay

to a `RTCPeerConnection` object before making a WebRTC media call. We also report additional time to display local streams on the browsers. This time corresponds to first firing of `canplaythrough` event from html video player. Experiments show that performance of capturing video streams depends highly on the camera in use and its driver implementations. Chrome can capture streams almost instantly, but the time it takes to present the streams locally is comparable to Firefox.

In WebRTC peer connection establishment, peers prepare call offers and answers with a description of call contents. Additionally, peers perform ICE hole punching for finding each other. As can be seen in Figure 2, ICE hole punching and preparing call/answer offers are overlapped. Connection establishment depends on network quality, but analyzing network performance is not the goal of this benchmark. However, two different network configurations were considered in this set of experiments. This allows us to (1) obtain a relative WebRTC performance on a given fixed network and (2) compare connection establishment to the rest of the setup time. We measure the time taken from starting ICE hole punching by the first peer until ICE agents of both peers get connected. For this experiment, default STUN servers provided by browser vendors were used to gather browser public addresses.

Figure 6 depicts the time required for making offers/answers for both data and media contents. Figure 7 shows the average value over 10 experiments for a variety of devices and networks. Worst case observed for a local connection between two browser tabs for Google Chrome on a Desktop platform.

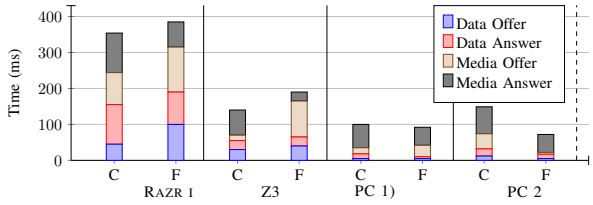


Fig. 6: Preparing call offer/answer delay overlapped with ICE hole punching

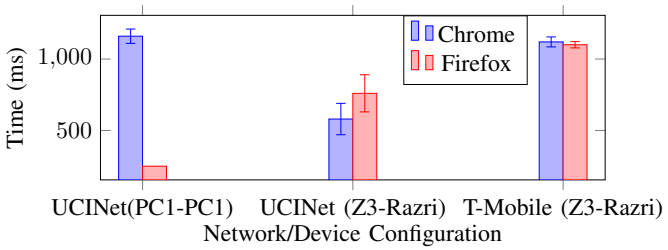


Fig. 7: ICE hole punching delay on different networks

B. Data Channel Performance

We conducted two experiments on datachannels in order to measure latency and throughput of data channels. Experiments are performed using two devices (Razr i and Z3) and are repeated on two different networks.

First experiment measures latency for delivering one SCTP user message[13]. Latency is especially crucial for having a responsive real time experience such as in multi player gaming. We measured it as the half of round trip time for one user message. This value reflects network latency and also data buffering policy used by implementations.

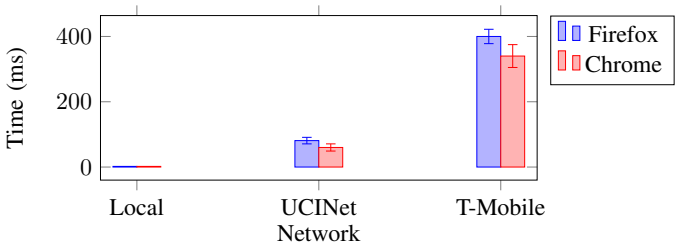
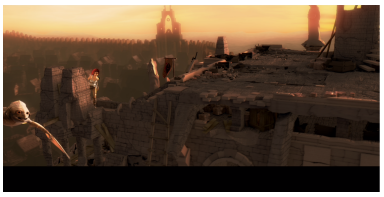


Fig. 8: RTT for Datachannel user messages

Next experiment measures transfer time for sending files over the networks using Datachannels. Figure 9 shows the transfer delay for files with different sizes sent from Device "Z3" to "Razr i". Note that Firefox browsers can send files with any size using DataChannel APIs, but if Chrome is used, files must be split into small chunks (64KB) first and sent separately. This procedure is done at the application level.

(a) Soccer



(b) Life

Fig. 11: Reference video inputs

TABLE VII: Reference video inputs specifications

Name	Resolution(s)	Frame rate	Duration
Soccer	640 × 480	60	20s(loop)
Sintel	1280 × 720	24	60s(loop)

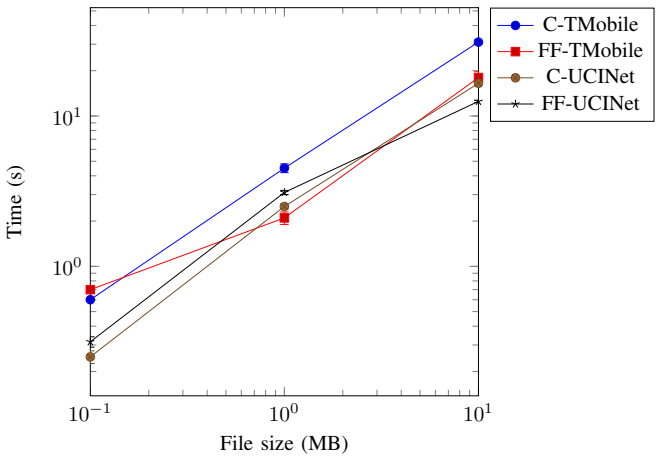
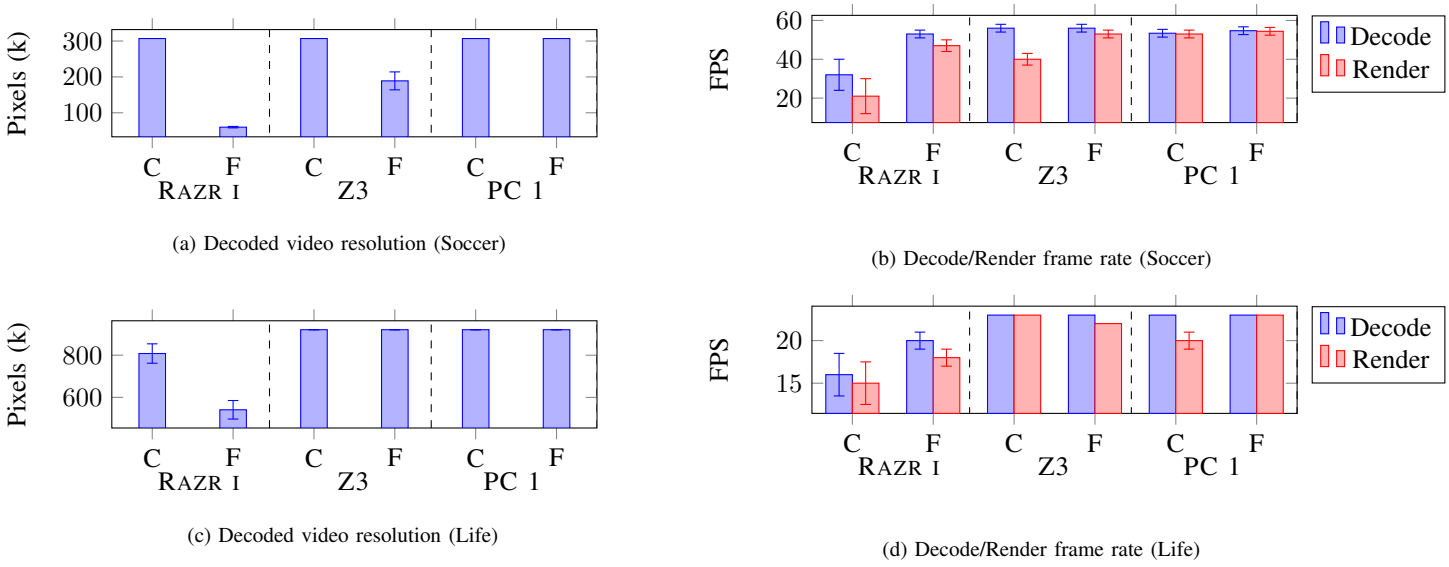


Fig. 9: Transfer time for files with different size

C. Media Engine Performance

Two experiments were conducted to evaluate media engine performance. Experiments are based on live videos captured by cameras as well as two video sequences in uncompressed Y420 format obtained from xiph.org archive (available at http://media.xiph.org/video/derf/) and shown in Figure 11.

The first experiment measures video encoder bitrate ramp up over time. Encoders increase output video bitrate gradually to avoid network congestion based on feedback from other peer until reaching the maximum pre-set value. Ramp-up rate in general depends on how well the other peer is



(a) Decoded video resolution (Soccer)

(b) Decode/Render frame rate (Soccer)

(c) Decoded video resolution (Life)

(d) Decode/Render frame rate (Life)

Fig. 10: VP8 decoder statistics

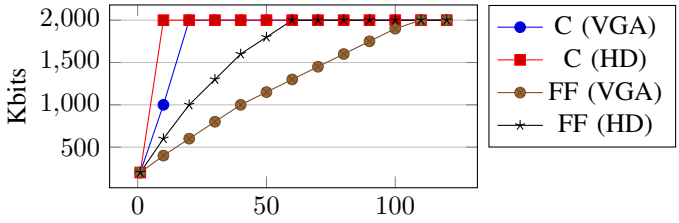


Fig. 12: Encoder bitrate over time

receiving/decoding frames. Since lower encoder bitrate will result in lower video quality, user experience may suffer during this period. Encoder performance is evaluated by recording the actual bitrate of encoded stream (a.k.a. sending bandwidth), encoding frame rate, and encoding resolution. On platforms with weaker CPU or hardware encoder, encoding bitrate, frame rate and resolution will drop below the pre-set value. Running this test requires a remote peer with enough decoding capability to rule out encoding quality drop due to decoding capability limitation. Figure 12 shows the encoder bitrate values for videos by different resolution captured by cameras attached PC1 to Z3 devices over the UCINet network. As shown, encoder ramp-up is not fast enough for Firefox implementation and it can hurt user experience.

Next experiment measures decoding performance of VP8 coded videos for different platforms and implementations. Running decoder performance test requires a remote peer with enough encoding capability so that input stream with given constraints can be guaranteed. Google chrome on Desktop is used to encode two reference videos from table VII. Additionally, `RTCPeerConnection` is established with a fixed video bandwidth constraint (2000Kbps) when creating an offer to remote peer. This constraint makes sure the average video bitrate transferred between peers does not exceed this value. In our experiments, we found 120 seconds to be enough for encoder to reach its peek bitrate. Hence, measurements start after two minutes and continue for one minute during which performance statistics are measured. Figure 10 depicts average decoded video resolutions along with decoding and render rates measured in the one minute interval for both reference videos. It shows that processing high definition videos is a challenging task for mobile phone devices. This condition can become even worse for multi point conversation where video are encoded/decoded concurrently per each participant. We see two different policies used by browsers to cope with limited resources: Google Chrome tends to keep the resolution suffering from lower frame rate, Mozilla Firefox scales down the video in order to keep the frame rate steady.

V. Conclusions and Future Works

This paper presented a benchmark for performance evaluation of WebRTC implementations across different platforms and architectures. Quick establishment of a RTC peer connection is an important requirement for future web applications. Also, good performance of media and data transfers especially on mobile devices, is extremely important for having a better user experience. These factors make a benchmark such as WebRTCBench for performance evaluation as well as for guiding performance optimizations. It is helpful for both industrial and end users. To demonstrate capabilities of our benchmark, initialization and peer connection establishment and data/media encoding/decoding measurements for variety of browsers and devices is presented. We plan to extend WebRTCBench by adding future WebRTC functionalities as they emerge in WebRTC implementations.

VI. Acknowledgments

This work was supported by the Intel Corporation. The authors would like to thank Mozilla engineers for their comments and the anonymous reviewers for their helpful suggestions. We also thank Vivek Krishnan for working on the initial version of benchmark.

REFERENCES

[1] Banabread, https://developer.mozilla.org/en-us/demos/detail/bananabread/, accessed: 10-10-2014.

[2] Media stream depth stream extension, https://www.w3.org/wiki/media_capture_depth_stream_extension, accessed: 4-14-2015.

[3] Octane javascript benchmark, https://developers.google.com/octane/. accessed: 4-14-2015.

[4] Statcounter, http://gs.statcounter.com/, accessed: 10-10-2014.

[5] I. Fette and A. Melnikov. The websocket protocol. 2011.

[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 20th International Packet Video Workshop*, pages 1–8, 2013.

[7] E. Ivov, E. Rescorla, and J. Uberti. Trickle ice: Incremental provisioning of candidates for the interactive connectivity establishment (ice) protocol. 2013.

[8] A. B. Johnston and D. C. Burnett. *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web*. Digital Codex LLC, 2012.

[9] J. Rosenberg. Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols. 2010.

[10] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session traversal utilities for nat (stun). 2008.

[11] V. Singh, A. A. Lozano, and J. Ott. Performance analysis of receive-side real-time congestion control for webrtc. In *Proceedings of the 20th International Packet Video Workshop*, pages 1–8, 2013.

[12] S. Tilkov and S. Vinoski. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010.

[13] M. Tuexen, S. Loreto, and R. Jesup. Webrtc data channels. 2015.

[14] A. Van Kesteren and D. Jackson. The xmlhttprequest object. *World Wide Web Consortium, Working Draft WD-XMLHttpRequest-20070618*, 2007.