Albert Abelló Lozano

Performance analysis of topologies for Web-based Real-Time Communication (WebRTC)

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 20.3.2012

Thesis supervisor:

Prof. Jörg Ott

Thesis advisor:

M.Sc. (Tech.) Varun Singh

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Albert Abelló Lozano

Title: Performance analysis of topologies for Web-based Real-Time Communication (WebRTC)

Date: 20.3.2012 Language: English Number of pages:6+38

Department of Communication and Networking

Professorship: Networking Technology Code: S-55

Supervisor: Prof. Jörg Ott

Advisor: M.Sc. (Tech.) Varun Singh

Your abstract in English. Try to keep the abstract short, approximately 100 words should be enough. Abstract explains your research topic, the methods you have used, and the results you obtained.

Keywords: Resistor, Resistance, Temperature

Preface

Thank you everybody.

Otaniemi, 9.3.2012

Albert Abelló Lozano

Contents

List of Figures

List of Tables

Definitions and abreviations

1 Introduction

The need of a new way to communicate between two points of the planet is a problem that many different technologies have tried to approach. Systems such Skype or VoIP are not able to cope the needs of the new generations of developers and users that everyday require a more integrated way of communication with the World Wide Web (WWW).

Besides this, the amount of data transferred during the last years and the prevision for the future allocates a new scenario where non-centralized systems such as P2P are required as data bandwidth grows and systems need to become more scalable. Nowadays, networks are still manly content-centric, meaning that data is provided from a source to a client in a triangle scheme, clients upload data to central servers and this data is transferred to the endpoint. This architecture has been provided since long time as reliable and scalable, but with the appearance of powerful applications and Video On Demand (VOD) scalability is becoming an issue.

Those circumstances lead to a whole new world of real-time browser based applications which require also a new framework to work with. Ranging from online videoconferencing to real-time data applications, for this purpose few attempts were made in the past being highly reliable on specific hardware and custom-built no-compatible systems. Those proposals were not accessible to normal users that could not afford to adapt the requirements.

All previous concepts are now possible thanks to the increase of the average performance in every computer nowadays, this situation is helping to build more complex browsers that are able to perform many different tasks that enhance web browsing to a different level. Having a browser to handle OpenGL style of applications is now possible thank to the new HyperText Markup Language version 5 (HTML5) standard. Multimedia abilities are also able to be reproduced on those browsers and webcam media shown as HTML is now a reality. Even dough, there is still an important issue that must be addressed: there is no common standardized protocol that allows developers to do this. Web Real-Time Communication (WebRTC) effort to approach this problem is to build a simple and standard solution for peer-to-peer browser communication in the HTML5 environment [1] .

Internet bandwidth capabilities helped to take the decision to start integrating peer-to-peer solutions in browsed based applications, this is due the year-by-year increase of user bandwidth connectivity during the last 10 years. Actual latency in the network is low enough to allow real-time applications to work resiliently in the browser. The amount of users being able to transfer at high speed has increased during last years (Figure 1), about 39% of users are able to download at speeds greater than 4Mbps being this a very good average speed for multimedia content [2].

Regarding the specs on the client side, recent surveys and statistics taken by the game manufacturer Steam prove that more than 61% of machines are carrying 1 to 4 gigabytes of RAM and nearly 90% of computers handle 2 to 4 core CPU with a 64 bit OS [3], this environment can easily handle media enhanced applications that require high performance for media encoding. WebRTC concept rests over multiple layers having the browser as an underlying application, a traditional browser allocates a

	Country	% Above 4 Mbps	QoQ Change	YoY Change
–	Global	39%	-2.8%	6.9%
1	South Korea	84%	-2.2%	28%
2	Switzerland	79%	2.3%	14%
3	Netherlands	79%	-5.0%	-2.0%
4	Japan	74%	2.4%	20%
5	Belgium	69%	-4.7%	-6.3%
6	Canada	69%	0.8%	8.2%
7	Hong Kong	68%	-5.2%	-4.7%
8	Latvia	67%	-1.7%	-2.8%
9	Denmark	66%	5.2%	5.0%
10	Czech Republic	64%	-6.4%	-10%
...				
13	United States	57%	-4.6%	7.9%

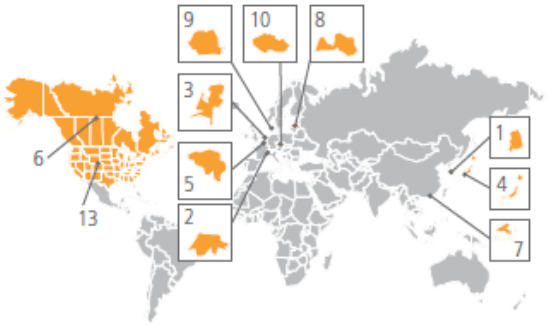

Figure 1: Broadband connectivity statistics about the speeds over 4Mbps around the globe.

lot of resources for running being the performance of the machine a bottleneck in some cases.

Traditionally, WebRTC concept approaches rely on the usage of plug-ins or other separate software components that make the system run smoother by avoiding one layer of processing (browser) but being non-standard and not cross-compatible, one of the most import ant concepts when designing applications nowadays. This approach has a new alternative with the arrival of the new HTML5 where WebRTC is integrated as one of the new Application Programming Interfaces (APIs) available alongside other many different interesting capabilities.

1.1 Background

WebRTC API is included into the HTML5, this is the fifth version of the WWW language. This version includes different API's and JavaScript codes that help the developer to easily introduce new features into their already existing WWW applications. The initial HTML version (2.0) was published in November 1995 with the only goal of delivering static content from the server to client browser [4]. HTML became de de facto format for serving web information.

HTML is written in tag formatting to identify different elements. Those tags are then interpreted by the browser to show the different data content served by the server. During the evolution of the WWW different new features have been added to the HTML standard and new versions where published, things like JavaScript and Style Sheets increase the flexibility and features of the WWW content enhancing the final user experience.

Due to the need to extend the features of the already existing HTML4 standard, a new version was proposed in 2004 by the Mozilla Foundation and Opera Software [5]. This new proposition focused in new developing technologies that could be backwards compatible with the already existing browsers, the idea didn't make a success and was tier apart until January 2008 when the first Public Working Draft was published by the Web HyperText Application Technology Working Group

(WHATWG) in the W3C [6].

This proposal had a greater reliance in modularity in order to move forward faster, this meant that some specs that were included in the initial draft moved to different working groups in the W3C. Those technologies defined in HTML5 are now in separate specifications, one of them being WebRTC. WebRTC works as an integrated API within the browser that is accessible using JavaScript and is used in conjunction with the Document Object Model (DOM) interfaces. Some of the APIs that have been developed are not part of the HTML5 W3C specification but are included into the WHATWG HTML specification.

1.2 Contribution

Investigate how WebRTC performs in a real environment trying to evaluate the best way to set multiple peer connections able to transfer media and data in different network topologies. Measure the performance of WebRTC in a real environment identifying bottlenecks related to encoding/decoding, media establishment or connection maintenance. All this should be performed in real-time over a browser by using the already existing WebRTC API.

Using metrics related to RTT, latency, packet loss and bandwidth usage we expect to understand the way WebRTC performs when handling multiple connections.

1.3 Goals

WebRTC uses and adapts some existing technologies for real-time communication. This thesis will focus in studying how:

- WebRTC performs considering different topologies using video/audio acquired form the Webcam using the API and encoded using different codec types provided by the standard.
- Usage of WebRTC to build a real application that can be used by final users proving that the API is ready to be deployed and is a good approach for the developer needs when building real-time applications over the web. This will be done in conjunction with other new APIs and technologies introduced with HTML5.

The final conclusion will cover an overall opinion and usage experience of WebRTC, providing some valuable feedback for the needs and requirements for further modifications on the API.

1.4 Structure

Not sure about here

2 Real-time Communication

Web Real-Time Communication is a technology that builds P2P applications by using a defined JavaScript API. The first announcement went public in a WG of the World Wide Web Consortium (W3C) in May 2011 [7] and started the official mailing list in April 2011 [8]. During the first stage of discussion, the main goal was to define a public draft for the version 1 API implementation and a route timeline with the goal to publish the first version by March 2013. The first public draft of W3C came public the 27th of October 2011 [9]. During this first W3C draft, only media (audio and video) could be sent over the network to other peers, it is focused in the way browsers are able to access the media devices without using any plugin or external software.

Alongside to the W3C working group, the WebRTC project also joined the IETF with a WG in May 2011 [10] with the first public announcement charter done the 3th of May 2011 [11]. Milestones of the WG initially marked December 2011 as deadline to provide the information and elements required to the W3C for the API design input. On the other side, the main goals of the WG covered the definition of the communication model, session management, security, NAT traversal solution, media formats, codec agreement and data transport [11].

One of the most important steps during the process of standardization came the 1st of June 2011 when Google publicly released the source code of their API implementation [12].

During all this period both WGs have been working alongside to provide a reliable solution to enable cross-platform applications to perform media and data P2P transfer over the browser in a plugin-free environment. The first final version of the WebRTC API is to be published at the end of 2013.

Some alternatives are available to the WebRTC concept, considering the global architecture of WebRTC, Session Initiation Protocol (SIP) and Secure Real-Time Media Flow Protocol (RTMFP) are similar approaches to the same solution.

2.1 SIP

Both SIP and RTMFP are protocols to allow communication between two different users with audio/video support. SIP is an open standard and RTMFP is a proprietary protocol by Adobe, both systems are widely used for real-time communication. SIP final Request for Comments (RFC) was published in June 2004, this chapter describes the methods and behaviors of SIP [13]. From an overview perspective, SIP is an application-layer control protocol for multimedia sessions, can establish, maintain and terminate them, during the development of the standard different new functionalities were added to the drafts such as conferencing and the possibility of adding/removing media from existing sessions. SIP differentiates from RTMFP/WebRTC by locating the end user to be used for communication, this feature allows SIP to be closely related to traditional PSTN networks as it allow cross-domain communication which is not possible when using RTMFP/WebRTC. SIP is not a complete toolkit for communications, it works alongside with other existing proto-

cols such as Real-time Transport Protocol (RTP), Real-Time Streaming Protocol (RSTP), Session Description Protocol (SDP) and Media Gateway Control Protocol (MEGACO). Using SDP for the session negotiation between the end-points and RTP/RSTP for the media transport, all those protocols usage is widely extended in the network and provides legacy for older technologies. Meanwhile SIP can locate and deliver a message to a user, SDP can provide the required information for the session establishment and RTP can transport the type of media specified in the SDP body.

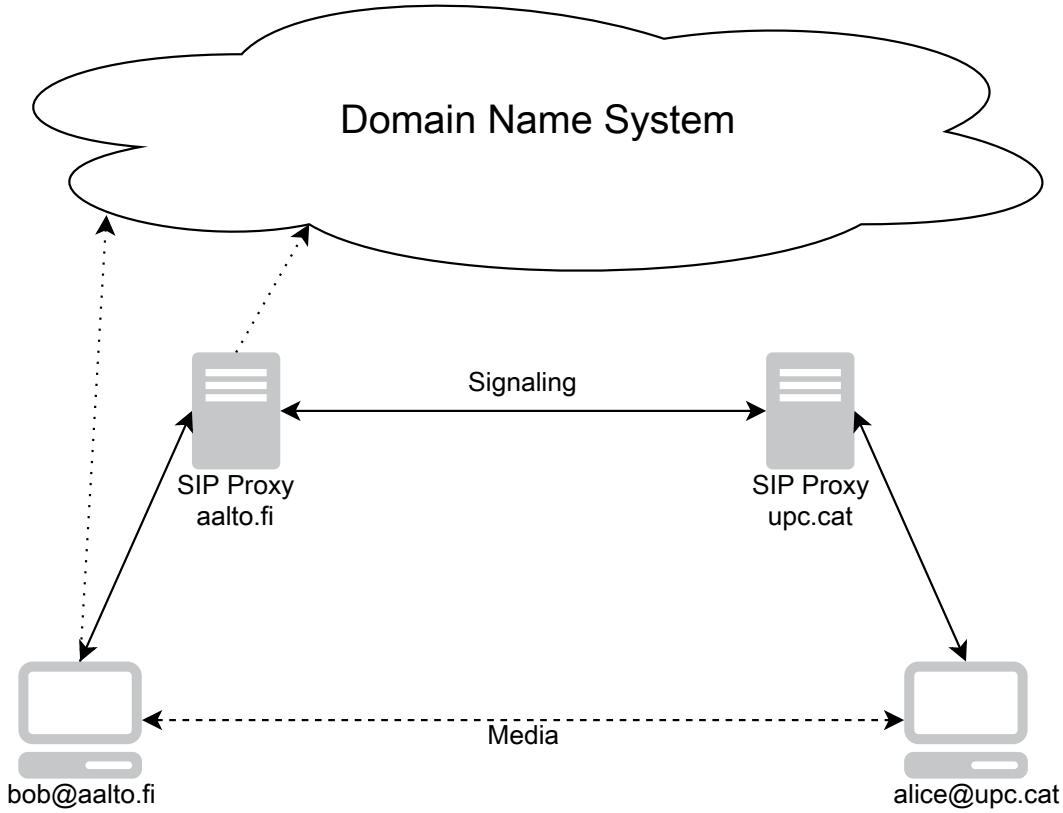

Figure 2: SIP architecture for end-to-end signaling.

SIP architecture relies in a trapezoid form where the Domain Name System (DNS) is used to locate the other peers of the system, once that peer is located and session is negotiated, media flows peer-to-peer directly to the endpoint. In order to build this system different agents are needed, SIP Proxies, SIP Redirect and SIP Registrar. SIP Proxies transmit the SDP and SIP messages from one peer to the other to establish communication (Figure 2). SIP Registrar are the machines that collect and save all the user information from the end points.

DNS provides the IP address for both proxy servers and allow the messages to be exchanged between both peers, when SIP is used the following messages are exchanged: INVITE, RINGING and 200OK. Those messages carry the SDP data inside in an object format, when ray@upc.cat receives the INVITE message from bob@aalto.fi builds the 200OK response carrying the SDP object that providing

compatibility check between both peers and which options and codecs to use. SIP provides some more messages to update the already existing session or to close them. The media transport is done using RTP and RTCP that rely over User Datagram Protocol (UDP) [13].

SIP is a pure Voice Over IP (VoIP) confederated technology that helped the community to learn about real-time P2P communication,we have used all the concepts and technologies embedded in SIP to build WebRTC.

2.2 RTMFP and Adobe Flash

RTMFP and Adobe Flash are proprietary technologies provided by Adobe, both services work together to provide multimedia and real-time communication between users.

Adobe Flash is a multimedia software that uses a plugin to work over the browser, it is used to build multimedia experiences for end users such as graphics, animation, games and Rich Internet Applications (RIAs). It is widely used to stream video or audio over webpages, in order to reproduce this content we need to install Adobe Flash plugin in our computer. It also carries different programming languages that drops from the standards called JavaScript Flash Language (JSFL) and Action-Stript. Due to the need of using a plugin his extension to tablets and mobile devices is more complicated than using standards. Adobe Flash Player is available in most platforms except iOS devices and reaches about 98 of all internet-enabled desktop devices. This plugin allows developers to access media streams from external devices such as cameras and microphones to be used along with RTMFP.

RTMFP uses Adobe Flash to provide media and data transfer between two end points. This system usually works over UDP [14]. Differing from SIP, this protocol is a full suite for media/data transfer in a P2P constraint environment and carries its own signaling methods and codecs. It also handles congestion control on the packets and NAT transversal issues. One of the biggest differences is that, similar to WebRTC, is not a full communication infrastructure and both peers must be in the same working domain to be able to communicate, is not a PSTN style of communication but a point to point messaging system. This protocol is implemented in Flash Player, Adobe Integrated Runtime (AIR) and Adobe Media Server (AMS), it is used for P2P communication between all those services [14].

This protocol is secured and encrypted, comparing with WebRTC, this issue has been addresses clearly in RTMFP by using proprietary algorithm and different methods. The RTMFP architecture is similar to WebRTC concept, it also allows reconnection in case of connectivity issues and works by multiplexing different media streams over the same media channel when handling conferences or multiple streams. For the signaling part Adobe uses a service called Cirrus (Figure 3), this service allows architectures such as: end-to-end, many-to-many and multicast [15].

Some of the most valuable features is the possibility to easy integrate P2P multicast topologies where one source sends a video to a group of receivers. This is something that none of the other protocols has implemented yet.

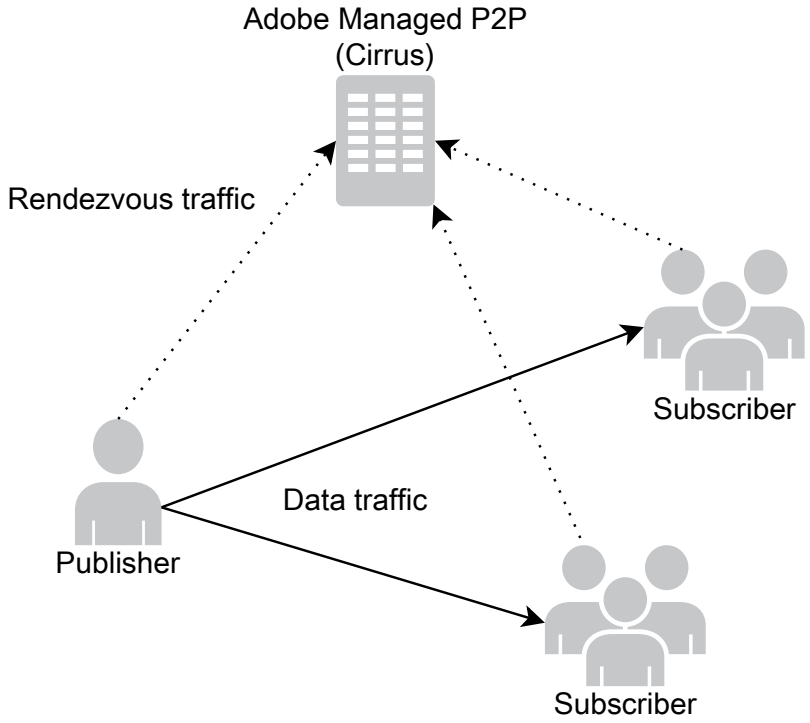

Figure 3: RTMFP architecture using Cirrus.

2.3 HTML5

WebRTC is part of the HTML5 package, both combined are an open cross-platform standard that aims to replace the Adobe proprietary proposal for P2P Real-Time Communication (RTC).

By using HTML5 features we avoid the need of installing any extra software to be able to use real-time multimedia applications on the browser.

2.3.1 Media Capture and Streams

HTML5 proposal will replace the existing need to use plugins when having multimedia features in web browsers. This new version carries different API that will be built into the browser and avoid the usage of external software to execute them, this scenario helps to build cross-platform standard applications in JavaScript instead of using plugins.

Compared with the existing Adobe Flash, APIs such as Web Graphics Library (WebGL) enables developer to build HTML5 3D and 2D games that will work in any browser natively without needing any especial software but with the rendering capabilities of Flash. Mobile browsers have been more likely to adopt this new technology for rendering [16]. The first final version of the API has been already published and browsers like Chrome and Firefox carry it.

Alongside with WebGL and many other APIs, this new HTML5 also carries the new Media Capture and Streams, also known as GetUserMedia API. This JavaScript

API allows developers to access local media such as video/audio from webcams and insert them in a web application by using the new video DOM element [17].

This proposal was first attached directly to the WebRTC group but has been published in a different draft, the usage of this API removes the need of using Adobe Flash to access the media device and also the plugin requirement. Developers can capture media streams from cameras and build them into Blob objects to be transmitted to other peers or reproduced locally.

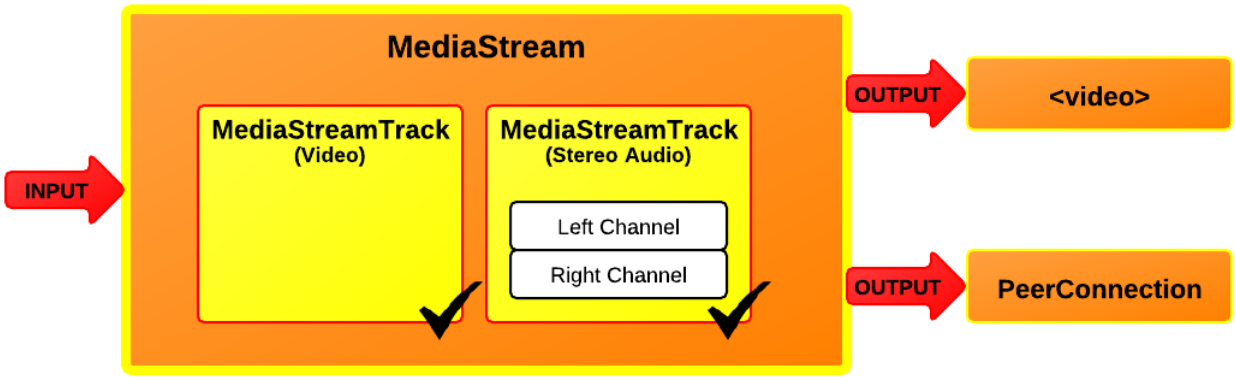

Figure 4: Media Stream API (source: W3C).

Figure 4 illustrates how the browser access that media and the outputs delivered to the developer. We will use this function to build WebRTC enabled applications for RTC video conferencing. The video tag is an HTML5 DOM element that reproduces local and remote streams.

2.4 WebRTC

WebRTC is an API part of the HTML5 proposal, it is defined in a W3C draft [7]. This API replaces the need of a RTMFP plugin for P2P communications for browsers, WebRTC uses already existing technologies, learned from SIP, bundled into an API. It is able to solve NAT transversal environments by using a mixtures of ICE, TURN and STUN technologies. For the session description it uses a modified bundled version of SDP. The format used for packet transport is RTP and SRTP, modified WebSockets are in use for P2P DataChannel implementation to provide data transport multiplexed over the same stream. All the traffic is sent over UDP or TCP over the same port [1].

This P2P session establishment system works in a constrained environment similar to RTMFP but it has been designed to provide legacy for other SDP based protocols such as SIP. It is a browser side API and does not provide any centralized service for signaling. Figure 5 shows how a WebRTC simple P2P scenario works, the server used for signaling is based in node.js.

Figure 5 does not show relay machines that provide NAT transversal solutions, those rely in other technologies that are applied to the API. In this simple example we consider both peers are in the same network without any Firewall or NAT restriction.

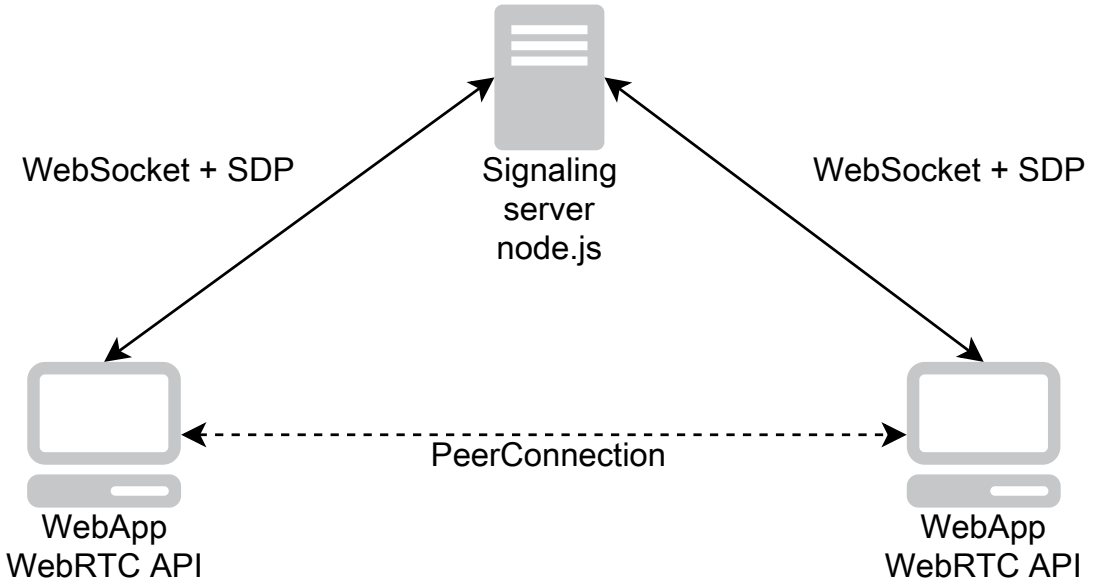

Figure 5: WebRTC simple topology for P2P communication.

The following companies and organizations have supported and are actively working in the development of WebRTC standard in the W3C: Google, Mozilla and Opera [18]. Other companies such as Microsoft have supported browser-to-browser solution but have published their own proposal which differs with the one published in the WebRTC WG, called CU-RTC-Web [19] which is a lower level API that claims to do everything that JSEP does.

During the firsts attempts to build a reliable solution for WebRTC Ericsson Labs presented an initial API based on the preliminary work done in the WHATWG, this API was called ConnectionPeer API and required an special module to be installed in your browser [20]. Ericsson lately dropped from the effort to build it's own browser to focus in the standardization and codec discussion, leaving the API implementation to the Mozilla and Chrome teams. The original API evolved rapidly during the next months thanks to the WGs and the developer community feedback that is experimenting with the unstable API.

During the process of standardization some important moments should be remarked. In January 2012 Opera implemented the first version of WebRTC getUserMedia for accessing the camera and audio [21], during this year getUserMedia is available in the stable version of Opera.

Google Chrome integrated the first version of WebRTC in its DEV and Canary channels of the browser during January 2012 [22], in June 2012 it started moving its API to the stable channel hidden behind a flag, in November 2012 WebRTC becomes fully available in Google Chrome stable channel and is open for public usage [23].

Mozilla Firefox started working on the getUserMedia implementation early 2012 delivering the first version of media access trough API at the beginning of 2012 in the alpha version [24], in April 2012 Mozilla published a WebRTC video demo running on Firefox in the "adler" channel [25], also supporting some primitive DataChannel

API. Later in October Firefox Nightly was carrying the first unstable version of the WebRTC API including DataChannel [26], Mozilla announced in September 2012 that the stable version of WebRTC will be shipped along with Firefox 18 in January 2013 [27], finally, the first public announcement of interoperability between Firefox and Chrome was done the 4th of February 2013 [28].

Some announcements done from Microsoft point out that they are also working in some implementation into Internet Explorer by using CU-RTC-Web as the default standard, at the moment only the Media API information is publicly available [29].

In October 2012 Ericsson announced the world's first WebRTC-enabled browser for mobile devices called "Bowser" with support for iOS and Android, this browser is able to handle WebRTC calls using RTCWeb Offer/Answer Protocol (ROAP) which is an old discontinued version of the WebRTC API that has moved to Javascript Session Establishment Protocol (JSEP). This browser also differs from the previous desktop alternatives on the codec side, it is carrying H.264 for video and G.711 for audio [30]. The API provided by Bowser is not fully W3C compliant.

2.5 Issues in WebRTC

WebRTC uses a mixture of different technologies to perform peer-to-peer communication between clients, those technologies range from SRTP, RTP, RTCP and multiple codecs that are being discussed. This scenario makes performance the key point for success in developing stable WebRTC applications.

Performance is manly related to computer capabilities and the ability to encode/decode at the same time as transferring and monitoring multiple peer connections. All those tasks are run over the browser and not directly on the OS, this is good for interoperability between platforms but bad in the performance aspect. Compared to Adobe technologies which uses a plugin, the performance they can deliver should be higher as they do not use as many application layers.

Media applications are delay sensitive and require a low packet loss for its proper function, WebRTC is working on this aspect by trying to implement congestion control over the connection stablished between peers, this work is not completed yet and will arise as a problem in the near future. Packet loss due to system capacity and bandwidth are measurable in WebRTC using the Stats API, this API provides information about the PeerConnection performance and is accessible by JavaScript.

Media constraints and bandwidth statistics will make a big difference in how media is acquired in WebRTC. Browsers and web applications have always tolerate some amount of delay and packet losses but this is not possible in media infrastructures for real time applications, an effort is needed to handle Quality of Service (QoS) in WebRTC to compete with RTMFP.

2.5.1 Quality of Service

Quality of Service (QoS) for WebRTC is being discussed and an internet draft is available with some proposals [31]. WebRTC uses DiffServ packet marking for QoS but this is not sufficient to help prevent congestion in some environments. When

using DiffServ the problem arises from the Internet Service Providers (ISPs) as they might be using their own packet marking with different DiffServ code-points, those won't be interoperability between ISPs, there is an ongoing proposal to build consistent code-points. Audio/video packets will be marked as priority using DSCP mappings with audio being more important than video or data [31].

The possibility to combine QoS in the transport layer with the constraints and stats of the WebRTC API will help developers to build more adaptive applications, for example, lowing the Frames per Second (FPS) in the case of high packet losses will reduce the bandwidth usage in the case of congestion of the link. This is possible thanks to the Stats API that provide the data statistics for the peer connection.

Some environments will also require better QoS as their bandwidth will be lower, examples in the use case draft relate this to surveillance cameras or similar approaches [32]. In these cases QoS should be modified by using the API, this situation can lead also to malicious JavaScript injection that could flood the path with packets.

2.6 Security concerns

In order to establish a call in WebRTC we use a web server for the signaling part, on the browser side we rely on built-in standardized JavaScript calling APIs which are used by the web server to establish the call between two peers [32]. Figure 5 represents the simple topology for a WebRTC call, even this system is similar to other provided VoIP services, the web server is able to move logic from the JavaScript in the browser giving total control to the server.

Obviously, this system poses a range of new security and privacy challenges different from traditional VoIP systems. It has to avoid malicious calling or having a call established without user knowledge, considering that those APIs are able to bypass Firewalls and NAT, Denial of Services (DoS) attacks are a threat.

Nowadays browsers continuously execute JavaScript codes from accessed web sites, this also includes malicious scripts, but in the case of WebRTC this could lead to a big privacy threat. In a WebRTC environment we consider the browser to be a trusted unit and the JavaScript provided by the server to be unknown as it can execute a variety of actions in the browser. At minimum, it must not be possible for arbitrary sites to initiate calls to arbitrary locations without user consent [33]. To approach this, the user must make the decision to allow a call (and the access to its webcam media) with previous knowledge of who is requesting the access, where the media is going or both.

The previous procedure is run by the JavaScript provided by the server, this is a security issue as the user must trust an unknown authority server. Calling services commonly use HTTPS for authentication whose origin can be verified and users should be verified cryptographically (DTLS-SRTP). Browser peers should be authorized before starting the media flow, even this can be done by the PeerConnection itself by using some Identity Provider (IdP) that supports OpenID or BrowserID to demonstrate their identity [34]. Usually this problem is not particularly important in a closed domain, cases where both peers are in the same social network and pro-

vide their profiles to the system and those are exchanged previous to the call, but it arises as a big issue when having federated calls from different domains such in Figure 6.

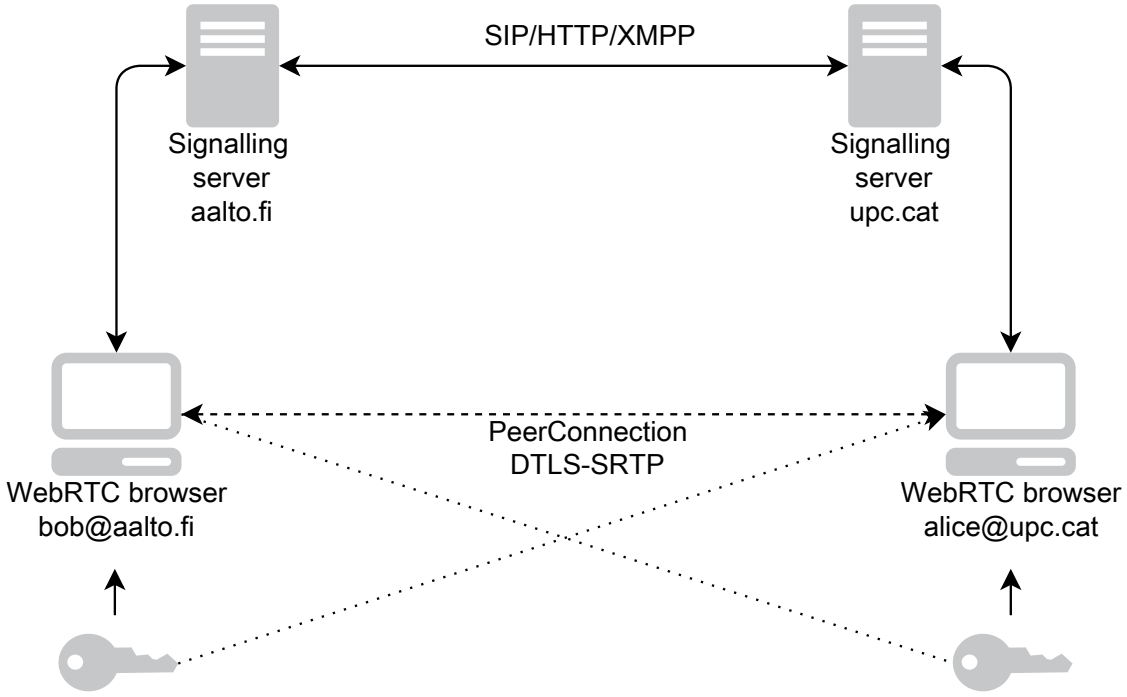

Figure 6: WebRTC cross-domain call with Identity Provider authentication.

If the web service is running over a trusted HTTPS certificate and has been authenticated it will be possible for the user to set the allow always access to the media, otherwise the user will have to allow this access. Once the media is acquired the actual API builds the ICE candidates for media verification. Authentication and verification in WebRTC is an ongoing discussion in the WG.

3 Topologies

We will talk about the different topologies that will be studied in this document that use WebRTC.

3.1 Point-to-Point

Point-to-point environments are widely used as a private calling system between two individuals, when related to WebRTC those use cases can be extended to people in the same domain so it is not required to be as private as other technologies.

Support systems are being considered as a valuable scenario for WebRTC point-to-point calls, this method will be useful as WebRTC does not need any plugin or account setup if we want an easy way to provide support to our customers in an HTML5 enabled site.

Other similar uses could be communication between doctor and patient in a medical application that intends to be cross-platform compatible in an ongoing standard. Communication in other scenarios such as citizens and authorities could also rely in a WebRTC application.

3.1.1 Challenges

In this scenario most challenges will come from the networking aspect. We could consider the enterprise use case, we will have ongoing calls between workers in the same enterprise network, this network could be restrictive to the use of non-trusted STUN servers or it might drop the UDP packets. Having restrictive NATs or Firewalls will directly affect the possibilities of having the WebRTC call correctly established, in this situation the possibilities of success highly rely on the network used.

When having point-to-point calls the problem that could arise from the local performance on the machine is not that critical considering that only one peer connection will be handled carrying two to three streams. We will focus on the user experience when having high restrictions in the NAT/FW.

3.2 One-to-Many

One-to-many scenarios are widely known as a type of multicast, one source sends the media to the different clients that connect to the origin. When having this topology the common uses rely on video and audio streaming to multiple clients, TV channels and streaming conferences are popular.

For example, we could have a major sport even being retransmitted to the viewers by using one-to-many. Other solutions could cover the use of WebRTC to have a CEO talking to the employees by using an HMTL5 web application. Music bands also could take advantage of this scenario by being able to transmit his show to the audience.

3.2.1 Challenges

In this scenario we will have a video, audio and data streaming connection from one source to multiple devices. This will cause a huge load on the source when having multiple PeerConnections running, local performance will be a constraint. Considering other scenarios, in this case, latency on the network is not as important as the rest due to the one-way communication only, no video and audio is needed to be received on the source.

We will need to study ICE and STUN mechanisms and how they perform in this scenario but the problems will arise from the source capacity on the hardware and link. Bandwidth demand on the source will be high and may affect the communication. On the other hand, having the audio delayed a couple of seconds is not going to affect the user experience in the call.

From the client perspective, the PeerConnection stablished will be easy to handle as no RTP streams will be sent back to the source, except the RTCP messages.

3.3 Many-to-Many

Many-to-many topologies are used for conferencing environment, this topology is used in systems such as Skype or VoIP. Conferences are used in enterprises for long-distance communication between employees and working groups, by this, the need of having those calls working smoothly for all participants is very important.

Due to the compatibility with HTML5 and the DataChannel spec that is shipped with WebRTC many-to-many environment could be also used for data transmission between different peers in a torrent scenario. Combining it with WebGL it could even be used for gaming experiences or file sharing.

3.3.1 MCU

MCU usage will surely be an option when designing WebRTC infrastructures, the ability to multiplex different streams into the same channel will directly affect on how the client performs when reproducing the video reducing the amount of used resources.

3.3.2 Challenges

When running multiple peer connections browsers on the client side will be obliged to handle multiple incoming streams and keep up all the connections stablished, this will be a problem regarding to the resources on the client side.

So, alongside with other problems mentioned in previous topologies, resources will directly affect on how this scenario performs in different users.

On the other side, even considering that NAT and Firewall connectivity issues are solved, we should be careful when guaranteeing all peer connections to be correctly stablished. TURN/STUN failures might directly affect to the success ratio in this topology.

4 Performance Metrics for WebRTC

This section will define the way we measure the performance in WebRTC environments, this real-time media environment will require an specific approach and some metrics to define how the protocol behaves in different topologies and scenarios.

Different issues might affect directly how the WebRTC media performs, these range from the hardware of the clients to the state of the link. In the following chapters we will describe some of them that will be used in our study cases.

4.1 Losses

Loss rate indicates packet losses during the transmission or processing. Usually packet losses affect directly the performance of a call and can indicate how the link is behaving between the different peers, in our case, packet loss will be a direct indicator of the quality of the ongoing WebRTC transmission. However, the packet loss indicate that some packets are not arriving, another strong indicator that goes attached is delay as packets will arrive later prior to getting lost in the link. This indicator will show up when the link is carrying big congestion of failures.

Some delayed packets should also be considered as losses as they won't be useful anymore for the ongoing connection, those packets won't show up in the stats as losses. In WebRTC loss rate will affect directly to the ongoing transmission as the delay range that we can tolerate is very low before the quality of the call deteriorates, some data-driven WebRTC connections will tolerate some more delay. In general case Loss Rate will be considered as a main point for recalculating the path by using faster routes. This indicator is manly attached to link quality.

Losses will be calculated in a certain period of time so we will be able to see how much loss rate we have in a certain range of time.

$$\frac{PKT_{loss}(T) - PKT_{loss}(T-1)}{PKT_{received}(T) - PKT_{received}(T-1) + PKT_{loss}(T) - PKT_{loss}(T-1)} \quad (1)$$

Equation 1 calculates the estimated packet loss we might have on the link. This operation will be done every period, we will determine this period when building the testing environment.

4.2 Round-Trip Time (RTT)

The delay in a link can be measured form different perspectives, one-way delay indicates the time it takes for a packet to move from one peer to the other peer, this time includes different delays that are given in the link. This one-way delay is calculated form the time taken to process it in both sides (building and decoding), the lower layer delay in the client (interface and intra-layering delay), queuing delay (from the multiple buffers in the path) and propagation delay (speed of light). The sum of all those delays compose the total one-way delay.

Considering the structure of WebRTC, one of the most important delays that we will have to consider and study is the processing delay as our applications will rely

in a multiple layer structure, running over the browser will affect the performance compared to other technologies that run directly over the OS. Delays in our case will be symmetric as we will be sending and receiving media, the delay will be important in order to reproduce the streams in the best quality possible and avoid decoding artifacts in the media.

RTT will be an early indicator of congestion in a WebRTC connection, this RTT must be monitored and most important, the adequate RTT have to be defined for every connection as the clients won't be aware of the appropriate amount for good performance.

4.3 Throughput

Throughput will be a key metric for testing the performance of WebRTC environments, this value will show how much capacity of the link is taking each PC and stream. It is complex though as there is still no QoS implemented in WebRTC. The throughput metric is going to provide bandwidth for video/audio in each direction, we can then use this value to provide some quality metric averaging all the previous mentioned measures in order to monitor the overall quality of the call. A sudden drop of the throughput will mean that the bandwidth available for that PC has been drastically reduced, this will lead to artifacts, or in the word case, loose of communication between peers. In this specific situation ICE candidates will try to be renegotiated in order to obtain a different solution for the connection and reestablish the media with the best throughput possible.

4.3.1 Audio streams

When using real time media environments for bidirectional communication the user experience is a key indicator of success. One of the factors that have to be considered is the Noise Reduction (NR) and Acoustic Echo Canceler (AEC). Those mechanisms allow the call to be smooth and avoid extra noises and echoes from the speaker voice to be transmitted, in WebRTC will provide a strange behavior when measuring the throughput, when the is no speech the bytes transferred will be approximately zero, being the throughput negligible. This helps to reduce the bandwidth usage and provides a more comfortable conversation when having a call.

4.4 Other metrics

Besides the metrics explained in the previous sections we are keeping other important values that affect WebRTC.

CPU/RAM usages are logged in order to determine how an average system performs when running the different scenarios as some of them will be more demanding than others, this will give an approximate approach to the required resources needed.

Also call setup time and frequency of call drops will be saved, it might be important to determine an approximate call setup time since the start of the negotiation until the media arrives. By doing this, we are measuring a parameter that directly

affects the user experience in WebRTC. From the other side, call drops will be counted to see the call success ratio in every scenario.

We will also calculate and pay attention to the delay variation, this is important as it affects how the user interacts with the other peer during a call. Having high delay variations led to an uncomfortable call and distortion. We will measure this variation and the amount of delay that different topologies produce.

5 Evaluation Environment

We have set up a testing environment to help us run tests for WebRTC. Figure 7 describes the functional blocks used for the simple video call.

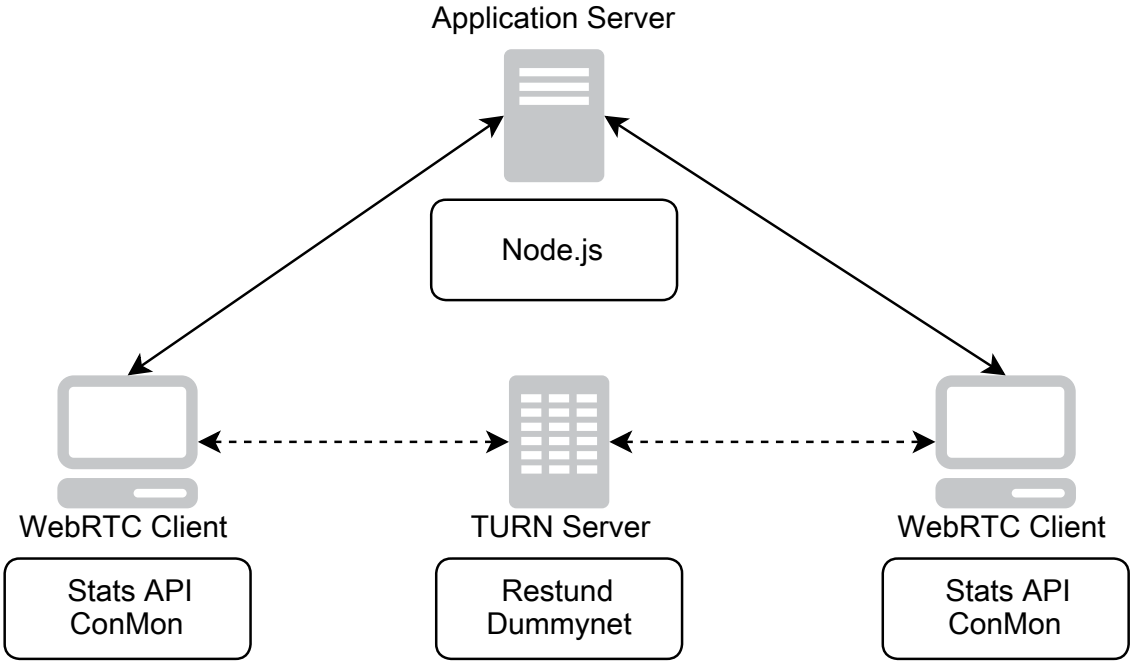

Figure 7: Description of testing environment topology.

5.1 WebRTC client

WebRTC clients are virtual machines that run a lightweight version of Ubuntu (Lubuntu) with 2GB of RAM and one CPU. This light version avoids the usage of 3D acceleration helping us to get better results in performance than compared with other distributions.

Clients will be running Chrome Dev version 27.01453.12 as WebRTC capable browser. To avoid modified results due to a bug in the *Pulse Audio* module of Ubuntu that controls audio input in WebRTC calls will be done with only video, the amount of audio transferred due to the echo cancelation systems can be neglicted.

5.1.1 Connection Monitor

Connection Monitor (*ConMon*) is a command line utility that relies on the transport layer and uses TCPDUMP to sniff all the packets that to go a certain interface and port [35]. This application is designed to specifically detect and capture RTP/UDP packets, relies on *libcap* for the capture in the network layer. This software detects and saves the header but discards the payload of the packet keeping the information we need for calculating our KPIs.

Typically we will run the PeerConnections between two devices and start capturing those packets by using *ConMon*. The PeerConnection will carry real data so the environment for testing will be a precise approach to a real scenario of WebRTC usage.

ConMon captures will be saved into different files and allow us to plot every stream bandwidth and calculate other parameters such as delay by using some parsing, this will allow us to compare how precise are both way of analyzing WebRTC as *ConMon* is working directly over the incoming interface and avoids all the processing that the browser is doing to send the stats to the JavaScript layer. Figure 8 represents one video stream from the same call as Figure 9 and **??** but captured from the *ConMon* application.

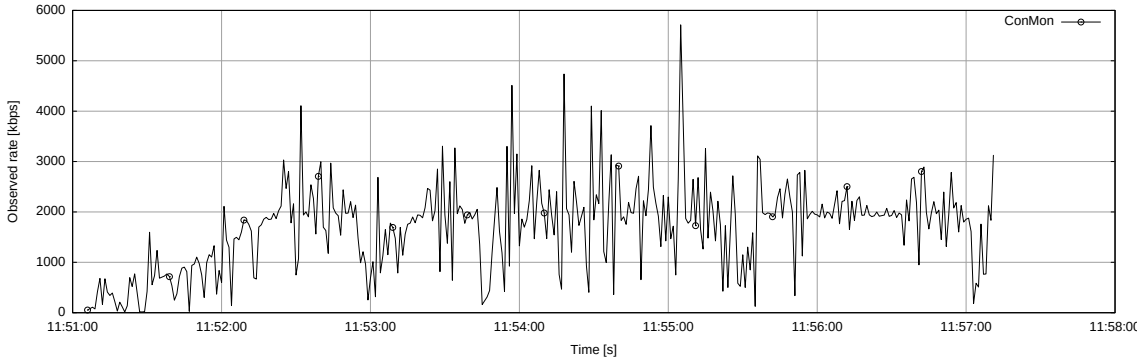

Figure 8: Point-to-point WebRTC video stream throughput graph using ConMon over public WiFi.

The capture from *ConMon* will be very accurate capturing all the packets that go through the interface, dumping the values into the output file, this data will then be processed and averaged for every second prior plotting. This processing will lead to some fluctuations on the graph that distort the reality.

5.1.2 Stats API

WebRTC carries a subsection of methods to help developers to access the lower layer network information, this methods return all different types of statistics and performance indicators that we will be using to build our own JavaScript Stats API. When using those statistics we will measure all the congestion KPIs to analyze them.

The method used is the RTCStatsCallback returns a dictionary object (JSON) that has be parsed and manipulated to get the correct indicators, this object returns as many streams as available in a PeerConnection, usually audio and video. This data is provided by the lower layers of the network channel using the RTCP packets that come multiplexed in the RTP stream [36].

The Stats API is the way that WebRTC allows the developer to access different metrics, as this is still in an ongoing discussion the stats report object has not been totally defined and can slightly change, the methods used by the Stats API are available on the W3C editors draft [37].

We have built a JavaScript tool that uses those stats from the browser to calculate the RTT, throughput and loss rate for the different streams that are being received. Those stats can later be saved into a file or sent as a JSON object to a centralized monitoring system. Our JavaScript grabs any PeerConnection passed through the variable and starts looping an iteration to collect those stats and either plot them or save them into an array for post-processing. Figure 9 shows an example capture of a call between two browsers in two different machines, Mac and Ubuntu, the call was made over Wifi open network with no firewall in the middle but with real traffic. The measures are directly obtained from the Stats API JS file we have built and post-processed using *gnuplot*.

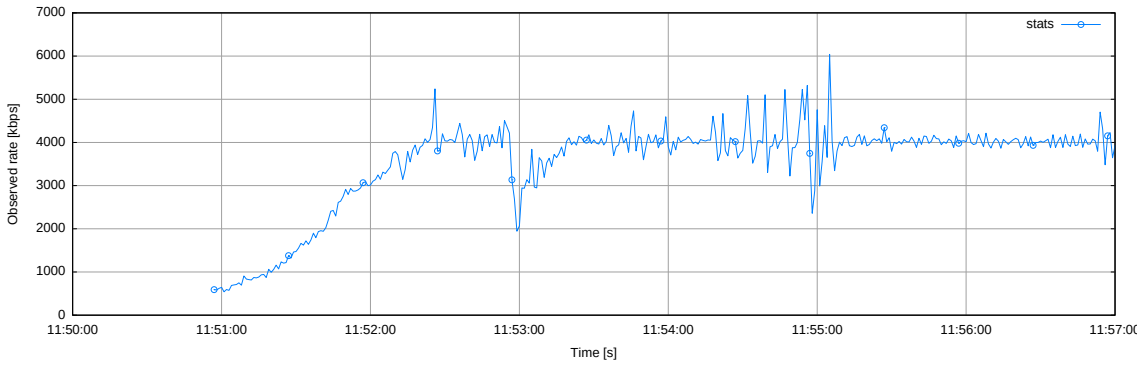

Figure 9: Point-to-point WebRTC video call total throughput graph using Stats API over public WiFi.

The previous Figure 9 considers the global bandwidth of the call, this means that the input/output video and audio are measured together to check how much bandwidth is being consumed over the duration of the call, as it is using RTCP packets for the metrics it takes a while to reach the average rate value. We can then plot all the different streams together to get an idea of how much bandwidth is consuming every PeerConnection.

5.1.3 Analysis of tools

Both tools will be measuring the same metrics but from different OS layers, this provides us some extra data to be considered in order to see how the our Stats API work and if it is possible to implement some extra features relying on that data for the WebRTC API.

Because of the period needed to measure the results it is possible to have strange behaviors when plotting the results as the information regarding to the next data period can be considered as the previous one. This is an accuracy problem that cannot be approached easily, when looking at the graph is important to see if both peaks (positive and negative) get compensated as this would mean that the data has not been allocated to the current period. This accuracy error is a problem that can be observed when comparing both *ConMon* and Stats API capture as the browser

will take some time to process the stats and send them to the JavaScript method, this will led to some extra error.

Figure 10 and 11 plot two video streams being captured from Stats API and *ConMon*.

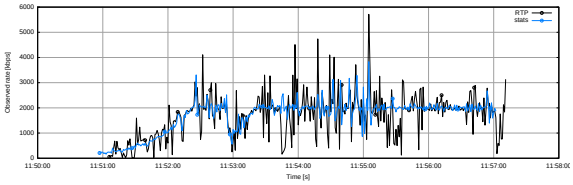

Figure 10: P2P incoming video stream comparison between ConMon and Stats API over public WiFi.

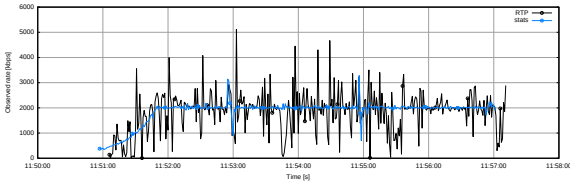

Figure 11: P2P outgoing video stream comparison between ConMon and Stats API over public WiFi.

Figure 10 represents the incoming media stream from the other peer, this is why the throughput seems to be so unstable in some parts of the call, consider also that this test was performed using wireless connection without any network conditioner. In Figure 11 local stream is sent from the peer capturing with *ConMon* to the remote peer, the throughput captured using Stats API will be much more stable around the 2000 Kbps.

5.2 Automated testing

For our testing scenario we have considered two options, manual and automated testing. The first test environment does not give as much accuracy due to the impossibility to iterate the test many times for the same configuration, if the second option is available the results can be averaged with all the iterations leading to an accurate result.

When considering both, the media being sent becomes a problem as there should be rich enough to be able to replicate a real call scenario. Google Chrome provides a fake video that can be activated by adding *–use-fake-device-for-media-stream* parameter, this video though might be too simple for our purposes.

Figure 12 represents the bandwidth that a real video call uses when sending the stream to the other peer, that capture shows the same stream from the origin an remote *StatsAPI* perspective. The bandwidth allocated goes up to 2000 Kbps. On the other hand, Figure 13 represents the same call by using the built-in fake video on both clients, the bandwidth in this case drops to an average of 250 Kbps. Those figures represent the same stream identified with the SSRC that corresponds, input from receiver and output from origin, this representation helps us to identify any possible distortion on the link. Google Chrome uses a bitmap system to draw the figures and components to be rendered in the video tag, this means that the amount of encoding and bandwidth used is low compared to a real webcam.

To address this issue in the video streamed from our automated devices we have built a fake input device on the virtual machines, procedure is described in Appendix A.

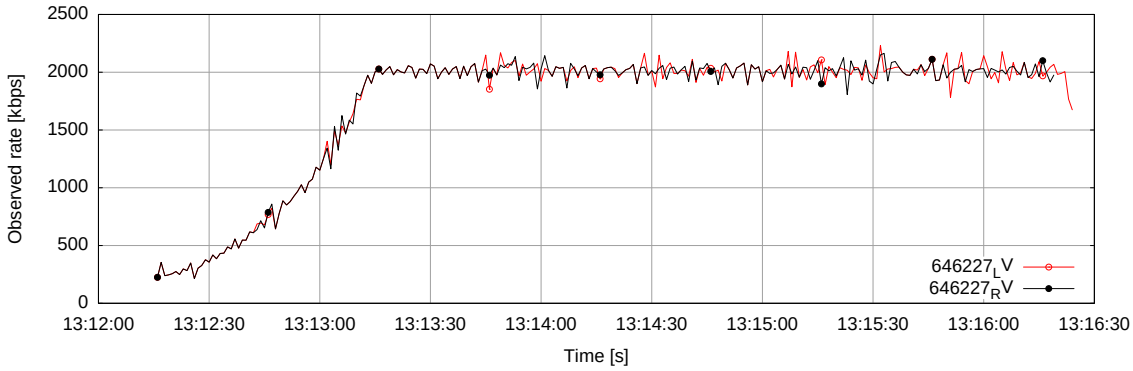

Figure 12: Video stream bandwidth using webcam input.

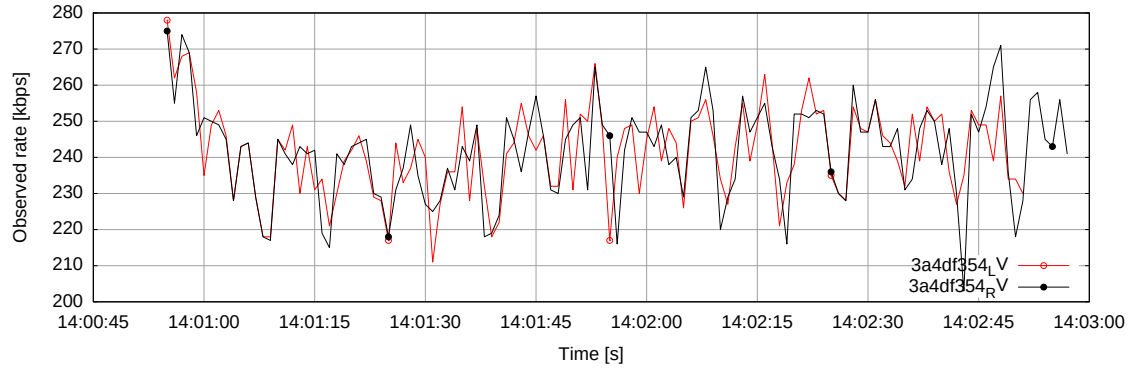

Figure 13: Video stream bandwidth using Chrome default fake video input.

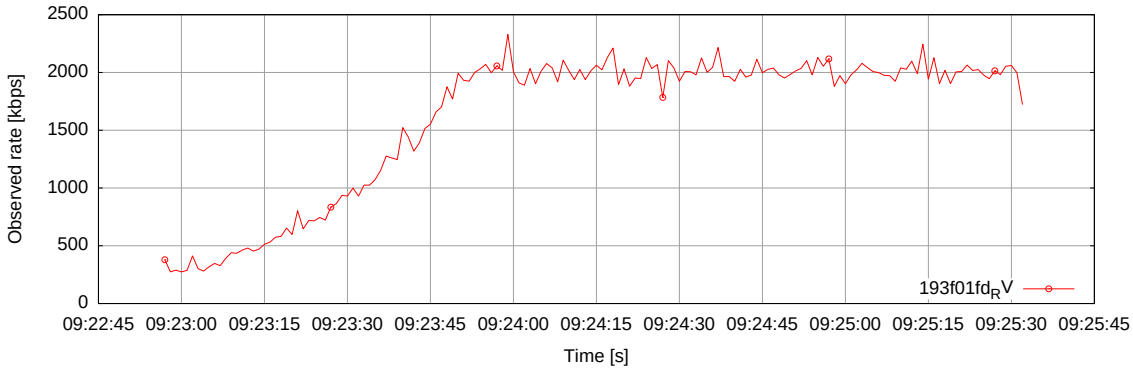

Figure 14: Video stream bandwidth using V4L2Loopback fake YUV file.

Figure 14 shows the bandwidth of a video stream measured by our *Stats API* using an YUV video captured from a Logitech HD Pro C910 as source, resolution is 640x480 at a frame-rate of 30 fps. Results show an approximate average bandwidth of 2000 Kbps which is a realistic approach to a real webcam. This setup will allow us to run multiple tests without the need of a webcam.

5.3 TURN Server

Our TURN server is used to pipe all the media as a relay to apply the constraints required for the tests, this machine is a Ubuntu Server 12.04 LTS with a tuned kernel to perform better with *Dummynet*.

As TURN software we are using *Restund* which has been proven to be reliable for our needs, this open source STUN/TURN server works with*MySQL* database authentication [**?**]. We have modified the source in order to have a hardcoded password making it easier for our needs.

To do so, we need to modify *db.c* file before compiling. Method *restund_get_ha1* content should be replaced with the following line of code where XXX is username and YYY the password.

```
md5_printf(ha1, "\%s:\%s:\%s", "XXX", "myrealm", "YYY");
```

This part is important as it allow us to set the constraints in a middle point without affecting the behavior of the WebRTC clients.

5.3.1 Dummynet

To check the performance of WebRTC we will need to modify the status of the network link. This is achieved using *Dummynet*, a command line network simulator that allow us to add bandwidth limitations, delays, packet losses and other distortions to the ongoing link.

Dummynet is an standard tool for some Linux distributions and OSX [38].

In order to get appropriate results with the constraints of the network we will have a machine acting as TURN for some tests, this machine will forward all the WebRTC traffic from one client to the other being transparent for both ends. The real goal of using TURN in WebRTC is to avoid and bypass some restrictive Firewalls that would block the connection, in our case, this works as a way to centralize the traffic flow through one path being able to be modified or tightened. From the performance perspective, when not adding any constraints to the TURN, the traffic and response is normal without the user noticing any difference.

Some problems arise when using *Dummynet* in our scenario, we will be using *VirtualBox* machines for some testing and to act as TURN, read Appendix B for more information about *Dummynet* configuration.

5.4 Application Server

Our application server will run the Node.js instance for the WebRTC signaling part, this machine runs Ubuntu with a domain specified as *dialogue.io*. This app is a group working application to allow people to chat and video call at the same time, we have modified it to build an specific instance for our tests, this instance will simply allow two users that access the page to automatically call each other and start running the JavaScript code with built-in *Stats API*

Most of this application is coded with JavaScript and use WebSocket protocol to carry the signaling messages.

6 Testing WebRTC

In this chapter we will study how WebRTC performs in different use cases and topologies previously described in chapter 3. All tests will be done using a real working environment with the tools previously mentioned in chapter 5.

6.1 Point-to-point

In a point-to-point scenario we have performed different tests to calculate how the application performs.

6.1.1 WiFi scenario

Firstly we have stablished a simple call between two peers that handle video and audio in an open WiFi network. This network does not carry any UDP packet filter or Firewall, the connection is performed without the need of STUN or TURN, we could easily say it is a straight forward peer-to-peer connection. The aim of this test is to observe how the captures differ between origin and receiver on the *StatsAPI* and *ConMon* layer.

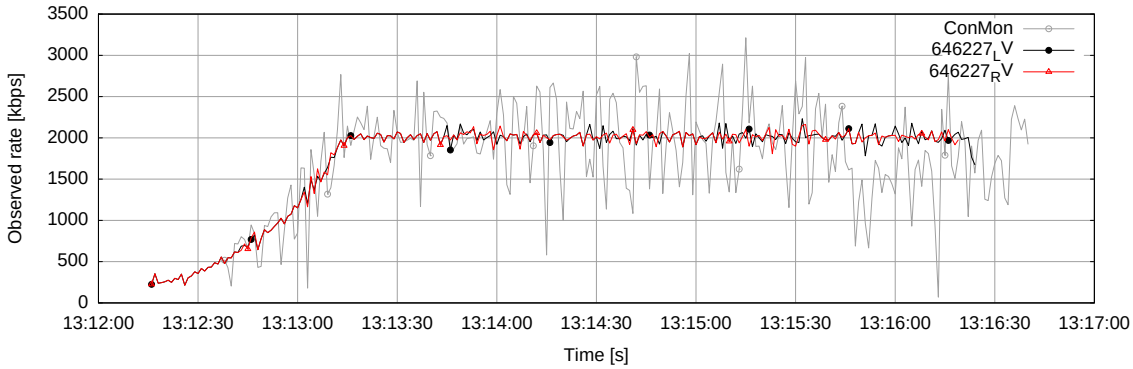

Figure 15: Point-to-point video stream plot using StatsAPI and ConMon data over WiFi.

Figure 15 represents the throughput rate on the same video stream, the three lines are the comparison between local video stream in origin peer, remote video stream in receiver peer and *ConMon* capture of the remote video stream on the receiver peer. All three streams contain the same data but they are measured in different layers, this will help us to understand the difference of throughput that is handling the overhead of the RTP and the disruption caused by the WiFi network.

Notice that red and black colors represent the Local Video (LV) and Remote Video (RV) from the same SSRC, both captures indicate the same stream captured using *StatsAPI*, and the grey line plots the capture performed using *ConMon* of the same SSRC. It is easy to observe that both *StatsAPI* captures are similar, some offset is produced due to the processing time between the network layer and the browser API that returns all values. Besides this, the capture is neat and throughput at the output of the origin client and input of the receiver is similar. Capture in the

network layer is more abrupt as all packets are captured and the period of calculus when plotting affects when the value is added, when having two opposite values peaks they should be balanced, meaning that the transmission in most of the period is stable and the peaks when plotting are a result of accuracy. Call duration in this test has been around five minutes. Some areas, mostly between 13.15.30 and 13.16.00, show a strange behavior of the link that might be produced by the WiFi, this throughput distortion is balanced on the WebRTC layer as the throughput delivered by the API does not change.

When we try to measure the quality of the call one important indicator is the delay, to calculate the delay we can either use the RTT measured by our *StatsAPI* or use the captures performed on the network layer by *ConMon*. The *ConMon* procedure will give us a high accuracy on the delay subtracting both timestamps from both of the clients, this will require to reduce the drift of the internal clock of the computers.

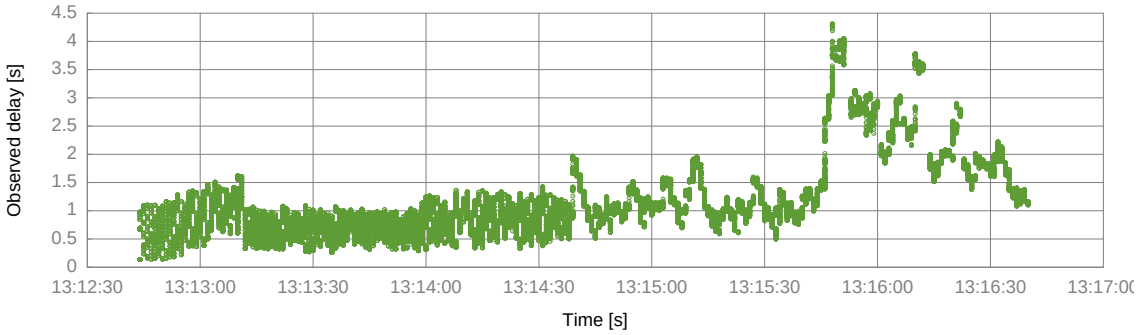

Figure 16: Delay calculated on the same stream captured using ConMon in both ends over WiFi.

Figure 16 represents the delay of the stream plotted in 15. We can see that the quality of the call is affected by the network distortion at the end of Figure 15, this variation of the throughput delivers a high delay of more than 4 seconds during some period of time between 13:15:30 and 13:16:30, the media received at that time will not render correctly and the user experience of the call is going to be worst than at the beginning of the call. A bursty WiFi network will led to delay even the bandwidth seems to be stable.

6.2 Non-constrained link test

After seeing how WebRTC performs in WiFi we are going to proceed with all tests in a controlled wired scenario adding different constraints to the link. This tests will be automated running ten iterations every time in order to get as much accurate results as possible.

Figure 17 plots the average bandwidth of every call in a wired network without any link condition, the average bandwidth obtained in the test is 1949.7 Kbit/s with 233 Kbit/s of deviation which gives the conclusion of having approximately 2 Mbit/s

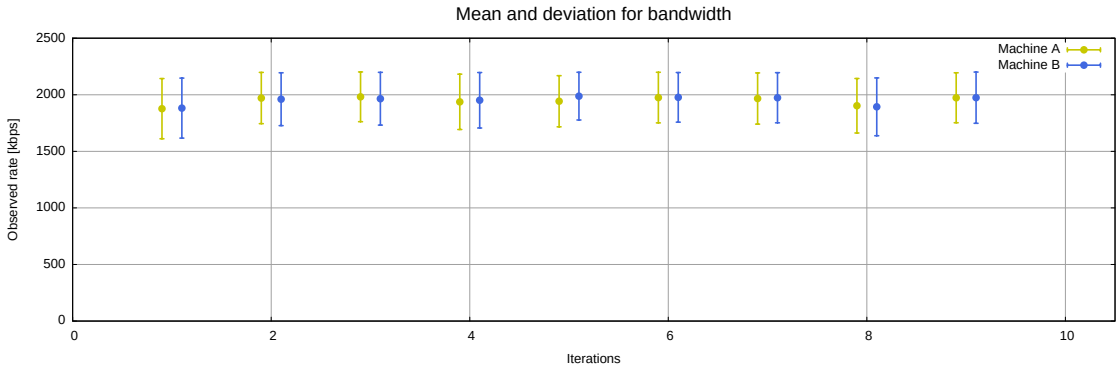

Figure 17: Bandwidth results for non-conditioned link.

	Machine A	*Machine B*	*Overall*
CPU (%)	48.76±2.76	48.83±2.78	48.79±2.77
Memory (%)	35.98±0.3	36.43±0.29	36.21±0.29
Banddwidth (Kbit/s)	1947.61±232.75	1951.76±234.5	1949.7±233.62
Setup time (ms)	1436.33±25	1447.44±22.71	1441.88±24.04
RTT (ms)	9.49±2.11	9.64±2.71	9.57±2.41
Delay (ms)	4.84±1.5	5.4±1.53	5.12±1.52
Losses (packets)	0	2	0.33
Failed call		2	

Table 1: P2P test with no link conditions.

standard bandwidth in a video stream for a non-conditioned link in WebRTC. Delay result in 5.1 ms with 1.5 ms deviation and RTT about 9.5 ms. Those results can be taken as standard for a non-conditioned WebRTC with high bandwidth resources. A summary of results is shown in Table 1. Some interesting results to track is the amount of calls failed in every test, considering all those calls go through a TURN server we might be able to approximate the success rate when establishing calls. All results go along with the deviation being this an important factor, in this test without any link conditioner we might have small deviation values such as milliseconds, but when adding conditions to the link those values will grow carrying less accuracy. Setup time is stablished as the time it take since the start of the PeerConnection object until the media stream from the other peer arrives, this value directly affects the time it takes for a user to be able to start talking, in the optimal environment it takes about 1.5 seconds to start the call.

Delay values in Table 1 are represented as a mean calculation of all the delay obtained in the link, thus this value is not representative of what happened in the call. Considering the example in Figure 16 we can see that the delay can variate during the call being the mean not appropriate to measure the response against the conditions of the link. In order to observe the behavior of WebRTC in delay we have two different approaches, the mean delay with deviation and delay distribution of

all calls.

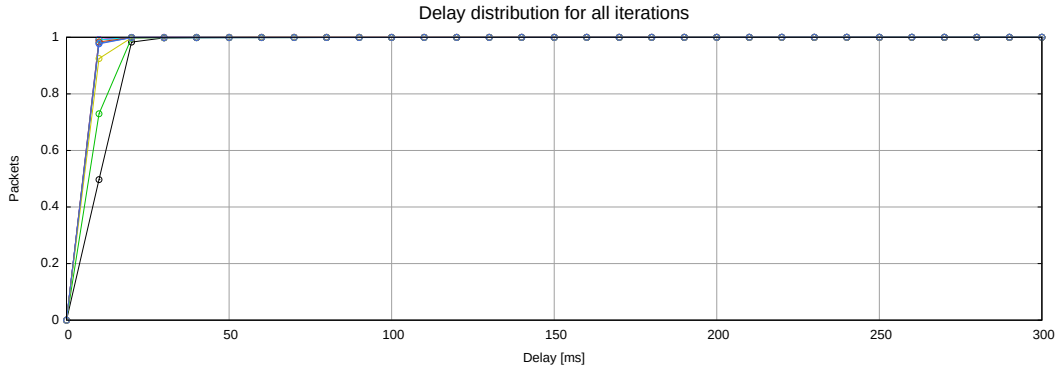

Figure 18: Delay distribution in each P2P iterations with no link constraints.

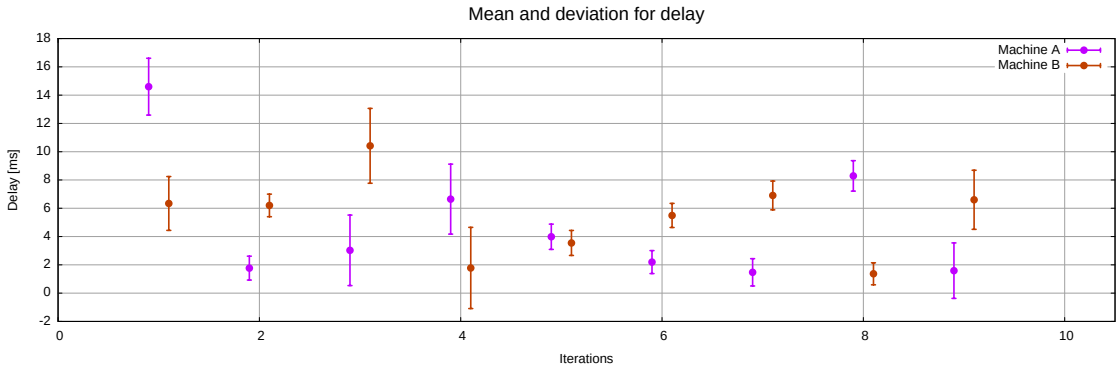

Figure 19: Mean and deviation for delay in each P2P iterations with no link constraints.

Figure 19 represents the mean and deviation of delay calculated for all iterations, this delay is calculated on basis with the arrival timestamp for each packet with the captures performed in both sides by *ConMon*. We run an NTPD daemon to calculate the drift on the time and sync both machines. There is small amount of drift of maximum 3ms in the worst case and as small as 1ms in the best one. In Figure 18, the distribution is given by the amount of packets that have some specific amount of delay, they are counted by batches of 10ms with a maximum range of 300ms. Most of the packets run with less than 25ms delay in all the iterations. The user experience with this small amount of delay with no aggressive steps in the plot will be barely negligible. Figures 19 and 18 differentiate from Figure 16 in the measurement of a global delay for an specific constraint scenario instead of just plotting a single call case, many aspects may affect the delay and the an optimal way to observe it is to plot the distribution and deviation of each iteration and try to guess a patron that repeats, Figure 16 is good to observe just one call if we add some conditions to the link meanwhile the call is going on.

	Machine A	*Machine B*	*Overall*
1% (Kbit/s)	1913.59±252.11	1880.24±261.46	1896.91±256.78
5% (Kbit/s)	1609.65±158.46	1527.84±198.59	1568.74±178.52
10% (Kbit/s)	1166.70±145.96	1114.94±177.88	1140.82±161.92
20% (Kbit/s)	333.34±65.99	295.46±57.98	314.4±61.98

Table 2: Averaged bandwidth with different packet loss conditions.

6.3 Behavior in lossy environments

We have performed some tests regarding lossy environments to see how WebRTC behaves in those, lossy situations can be given with some mobile environments with low coverage or just by having a busy link with no resources available.

We have tested the topology with 1, 5, 10 and 20% of packet loss, according to the results in Table 2 we are seeing a pretty good response from the internal algorithm up to 5% with small effect to the bandwidth and delay. When running with 10% loss the bandwidth drops to an average of 1140.8 Kbit/s and 162 Kbit/s deviation which is half of the corresponding amount for an standard call, this affects the quality of the link and video, 20% loss will affect to the performance dropping the bandwidth to an average of 314.4 Kbit/s with 62 Kbit/s deviation. We can say that the video quality will be worst with lossy networks but the delay is not affected, having a delay distribution response that matches the standard case without affecting the way users will talk, quality will be worst but the call will be correct in terms of usage. All metrics are in the normal range except bandwidth.

The algorithm used in WebRTC regarding to packet loss is proven to work fine in lossy environments with the results obtained, but there is a big gap of performance in the 10% loss network compared to the results with 20%, it is obviously a big amount of packets but the response with 20% is significantly better than the one with 10%.

6.4 Delayed networks

Another interesting situation that are given in mobile environments and queued networks is delay, we have also tested the performance of WebRTC in those conditions. We have benchmarked tests in different one-way delays, 50, 100, 200 and 500ms. In our case, the RTT results should be multiplied by two.

We have noticed that the system performs badly when having even small delays up to 100ms. The response of WebRTC is to reduce the bandwidth by discarding packets, this means that the congestion control systems that act in those environments are not working correctly. On the other hand, delay output does behave correctly having a continuous delay of the according time configured in the constraints, there are no sudden increases of delay and the deviation in delay fits in the standard limits.

	Machine A	*Machine B*	*Overall*
50ms (Kbit/s)	1909.31±258.09	1917.81±251.62	1913.56±254.86
100ms (Kbit/s)	1516.07±263.43	1453.94±272.79	1485±268.11
200ms (Kbit/s)	503.71±116.45	617.92±142.69	560.82±129.57
500ms (Kbit/s)	303.58±59.22	207.77±32.48	255.67±45.85

Table 3: Summary of averaged bandwidth with different delay conditions.

Table 3 represents the bandwidth response to the delay conditions, it is interesting to see that the deviation with the biggest delay is smaller than expected. Only with 50ms the system will output a good quality call, when increasing delay the performance of the video will decrease. WebRTC uses VP8 codec which degrades gracefully the quality in packet loss and delay conditions but the response in this case should be better if the congestion mechanisms worked properly.

7 Conclusion

The end.

References

[1] H. Alvestrand. Overview: Real Time Protocols for Brower-based Applications. https://datatracker.ietf.org/doc/draft-ietf-rtcweb-overview/, 2012.

[2] Akamai. The State of the Internet, 2ND Quarter, 2012 Report, 2012.

[3] Steam. Steam Hardware and Software Survey. http://store.steampowered.com/hwsurvey, October 2012.

[4] T. Berners-Lee. HyperText Markup Language 2.0. http://tools.ietf.org/html/rfc1866, November 1995.

[5] Mozilla Foundation and Opera Software. Position Paper for the W3C Workshop on Web Applications and Compound Documents. http://www.w3.org/2004/04/webapps-cdf-ws/papers/opera.html, 2004.

[6] Ian Hickson and David Hyatt. HTML5. http://www.w3.org/TR/2008/WD-html5-20080122/, January 2008.

[7] Web Real-Time Communications Working Group. http://www.w3.org/2011/04/webrtc/, May 2011.

[8] Harald Alvestrand. Welcome to the list! https://www.khronos.org/registry/webgl/specs/1.0/, April 2011.

[9] Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, and Anant Narayanan. WebRTC 1.0: Real-time Communication Between Browsers. http://www.w3.org/TR/2011/WD-webrtc-20111027/, October 2011.

[10] Real-Time Communication in WEB-browsers. http://tools.ietf.org/wg/rtcweb/, May 2011.

[11] Magnus Westerlund, Cullen Jennings, and Ted Hardie. Real-Time Communication in WEB-browsers charter. http://tools.ietf.org/wg/rtcweb/charters?item=charter-rtcweb-2011-05-03.txt, May 2011.

[12] Google release of WebRTC source code. http://lists.w3.org/Archives/Public/public-webrtc/2011May/0022.html, June 2011.

[13] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Ssession Iinitiation Protocol. http://www.ietf.org/rfc/rfc3261.txt, June 2004.

[14] M. Thornburgh. Adobe Secure Real-Time Media Flow Protocol. http://tools.ietf.org/html/draft-thornburgh-adobe-rtmfp, February 2013.

[15] Adobe. Cirrus FAQ. http://labs.adobe.com/wiki/index.php/Cirrus:FAQ, 2012.

[16] Chris Marrin. WebGL Specification. https://www.khronos.org/registry/webgl/specs/latest/, January 2013.

[17] Daniel C. Burnett, Adam Bergkvist, Cullen Jennings, and Anant Narayanan. Media Ccapture and Streams. http://dev.w3.org/2011/webrtc/editor/getusermedia.html, December 2012.

[18] Rian Liebenberg and Jan Linden. Introducing WebRTC an open real-time communications project. http://lists.w3.org/Archives/Public/public-webrtc/2011Apr/0001.html, April 2011.

[19] Bernard Adoba and Martin Thomson. Customizable, Ubiquitous Real Time Communication over the Web (CU-RTC-Web). http://html5labs.interoperabilitybridges.com/cu-rtc-web/cu-rtc-web.htm, August 2012.

[20] Stefan Hakansson. Beyond HTML5: Peer-to-Peer Conversational Video. https://labs.ericsson.com/developer-community/blog/beyond-html5-peer-peer-conversational-video, January 2011.

[21] Bruce Lawson. getUserMedia: accessing the camera and privacy UI. http://dev.opera.com/articles/view/getusermedia-access-camera-privacy-ui/, January 2012.

[22] Niklas Enbom. Real-time Communications in Chrome. http://blog.chromium.org/2012/01/real-time-communications-in-chrome.html, January 2012.

[23] Serge Lachapelle. See you on the web! https://sites.google.com/site/webrtc/blog/seeyouontheweb, November 2012.

[24] Robert O'Callahan. MediaStreams Processing Demos. http://robert.ocallahan.org/2012/01/mediastreams-processing-demos.html, January 2012.

[25] Anant Narayanan. WebRTC efforts underway at Mozilla! https://hacks.mozilla.org/2012/04/webrtc-efforts-underway-at-mozilla/, April 2012.

[26] Anant Narayanan, Maire Reavy, Randell Jesup, and Rob Hawkes. Progress update on WebRTC for Firefox on desktop. https://hacks.mozilla.org/2012/11/progress-update-on-webrtc-for-firefox-on-desktop/, November 2012.

[27] Robert Nyman. Full WebRTC support is soon coming to a web browser near you! https://hacks.mozilla.org/2012/09/full-webrtc-support-is-soon-coming-to-a-web-browser-near-you/, September 2012.

[28] Serge Lachapelle. Firefox and Chrome interoperability achieved. http://www.webrtc.org/blog/firefoxandchromeinteropachieved, February 2013.

[29] Media Capture API. http://html5labs.interoperabilitybridges.com/prototypes/media-capture-api-(2nd-updated)/media-capture-api-(2nd-update)/info, March 2012.

[30] Stefan Alund. Bowser - The World First WebRTC-Enabled Mobile Browser. https://labs.ericsson.com/blog/bowser-the-world-s-first-webrtc-enabled-mobile-browser, October 2012.

[31] S. Dhesikan, D. Druta, P. Jones, and J. Polk. DSCP and other packet markings for RTCWeb QoS. http://tools.ietf.org/html/draft-ietf-rtcweb-qos-00, October 2012.

[32] C. Holmberg, S. Hakansson, and G. Eriksson. Web Real-Time Communication Use-cases and Requirements. http://tools.ietf.org/html/draft-ietf-rtcweb-use-cases-and-requirements, December 2012.

[33] E. Rescorla. Security Considerations for RTC-Web. http://tools.ietf.org/html/draft-ietf-rtcweb-security, January 2013.

[34] E. Rescorla. RTCWEB Security Architecture. http://tools.ietf.org/html/draft-ietf-rtcweb-security-arch, January 2013.

[35] V. Singh. Conmon: App for monitoring connections. http://vr000m.github.com/ConMon/, 2013.

[36] C. Perkins, M. Westerlund, and J. Ott. Web Real-Time Communication (WebRTC): Media Transport and Use of RTP. http://tools.ietf.org/html/draft-ietf-rtcweb-rtp-usage, October 2012.

[37] Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, and Anant Narayanan. WebRTC 1.0: Real-time Communication Between Browsers. http://dev.w3.org/2011/webrtc/editor/webrtc.html, January 2013.

[38] Marta Carbone and Luigi Rizzo. The dummynet project. http://info.iet.unipi.it/~luigi/dummynet/.

[39] Patrick Hglund. Broken PyAuto test: WebRTC Ignores Fake Webcams. https://code.google.com/p/chromium/issues/detail?id=142568, August 2012.

[40] Patrik Hglund. V4L2 File Player. https://code.google.com/p/webrtc/source/browse/trunk/src/test/linux/v4l2_file_player/?r=2446.

[41] Marta Carbone and Luigi Rizzo. Dummynet Revisited. http://info.iet.unipi.it/~luigi/papers/20091201-dummynet.pdf, November 2009.

[42] Kernel Timer Systems: Timer Wheel, Jiffies and HZ. http://elinux.org/Kernel_Timer_Systems, 2011.

A Setting up fake devices in Google Chrome

To address the issue in the video that is transferred from our automated devices we have built a fake input device on the virtual machines that will be fed with a RAW YUV video of different resolutions and quality. This device will be added by using a hacked version of the *V4L2Loopback* which derives from the *V4L* driver for Linux, the modified version of the *V4L2Loopback* builds two extra devices as Chrome is unable to read from the same reading/writing device for security reasons, one of them will be used to fed the video and the other one to read it [39].

Differences between standard driver and hacked version:

- Need to write a non-null value into the the bus information of the device, this is required as Chrome input needs to be named as a real device. When using Firefox this is not required but works as well.

```
strlcpy(cap->bus_info, "virtual", sizeof(cap->bus_info));
```

- Our driver will pair devices when they are generated, this will create one read device and one capture device. Everything written into */dev/video0* will be read from */dev/video1*.

```
cap->capabilities |= V4L2_CAP_VIDEO_OUTPUT | V4L2_CAP_VIDEO_CAPTURE;
```

We used the code provided by Patrik Hglund [39] for the *V4L2Loopback* hacked version.

```
# make && sudo make install
# sudo modprobe v4l2loopback devices=2
```

Now we should be able to see both devices in our system, next step is feeding the */dev/video1* with a YUV file. In order to do this we will use the *V4l2 File Player* [40], this player executes on top of *Gstreamer* but adds a loop functionality to the file allowing long calls to succeed. Sample videos can be obtained from a Network Systems Lab.[1]

```
# sudo apt-get install gstreamer0.10-plugins-bad libgstreamer0.10-dev
# make
# v4l2_file_player foreman_cif_short.yuv 352 288 /dev/video1 >& /dev/null
```

We can now open Google Chrome and check if the fake device is correctly working in any application that uses GetUserMedia API.

[1] http://nsl.cs.sfu.ca/wiki/index.php/Video_Library_and_Tools

B Modifying Dummynet for bandwidth requirments

Dummynet is the tool used to add constraints and simulate network conditions in our tests.

Besides this, *Dummynet* has been natively developed for *FreeBSD* platforms and the setup for *Linux* environments is sometimes not fully compatible. Our system runs with Ubuntu Server 12.10 with a 3.5.0 kernel version on top of VirtualBox, this system requires to modify some variables and code in order to achieve good test results.

The accuracy of an emulator is given by the level of detail in the model of the system and how closely the hardware and software can reproduce the timing computed by the model [41]. Considering that we are using standard Ubuntu images for our virtual machines we will need to modify the internal timer resolution of the kernel in order to get a closer approximation to reality, the default timer in a Linux kernel 2.6.13 and above is 250Hz [42], this value must be changed to 1000Hz in all machines that we intend to run *Dummynet*. The change of timing for the kernel requires a full recompile of itself. This change will reduce the timing error from 4ms (default) to 1ms. This change requires the kernel to be recompiled and might take some hours to complete.

Once the kernel timing is done we will need to compile the *Dummynet* code, the version we are using in our tests is 20120812, that can be obtained form the *Dummynet* project site [38].

We should try the code first and check if we are able to set queues to our defined pipes, this part is the one that might crash due to system incompatibilities with FreeBSD and old kernel versions of Linux. If we are unable we should then modify the following code in the *./ipfw/dummynet.c* file.

```
if (fs->flags & DN_QSIZE_BYTES) {
        size_t len;
        long limit;

        len = sizeof(limit);
        limit = XXX;
        if (sysctlbyname("net.inet.ip.dummynet.pipe_byte_limit", &limit,
            &len, NULL, 0) == -1)
                limit = 1024*1024;
        if (fs->qsize > limit)
                errx(EX_DATAERR, "queue size must be < \%ldB", limit);
} else {
        size_t len;
        long limit;

        len = sizeof(limit);
        limit = XXX;
        if (sysctlbyname("net.inet.ip.dummynet.pipe_slot_limit", &limit,
            &len, NULL, 0) == -1)
                limit = 100;
```

```
        if (fs->qsize > limit)
                errx(EX_DATAERR, "2 <= queue size <= \%ld", limit);
}
```

When doing this we are making the file compatible with systems that have compatibility problems with the *sysctlbyname* function, XXX should be the value of the queue maximum length in slots and Bytes. Slots are defined considering a maximum MTU size of 1500 Bytes.

By default, maximum queue size is set to 100 slots, this amount of slots is not designed for bandwidth demanding tests such as 10Mbit/s or similar. In order to modify this we will need to set a higher value according to the maximum we require. Once this is set we need to recompile *Dummynet* form the root directory of the download source code and follow the install instructions in the README file attached to the code.

Even we have allowed *Dummynet* to accept more than 100 slots we won't be able to configure them into the pipe even the shell does not complain with error. The next step is to modify the module variables set in the */sys/module/ipfw_ mod/parameters* folder, this folder simulates the *sysctl* global variables that we would have running *FreeBSD* instead of Linux.

We need to modify the files *pipe_ byte_ limit* and *pipe_ slot_ limit* according to the values set in the *dummynet.c* previously modified.

Last convenient step is to add ipfw_mod to the end of */etc/modules* file so *Dummynet* module will be loaded even time the system starts.

We can now set large queues according to our needs.