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:7+73

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

Abbreviatures

AEC	Acoustic Echo Canceler
AMS	Adobe Media Server
API	Application Programming Interface
DNS	Domain Name System
DOM	Document Object Model
DoS	Denial of Service
DTLS	Datagram Transport Layer Security
FPS	Frames per Second
HTML	HyperText Markup Language
HTTPS	Hypertext Transfer Protocol Secure
ICE	Interactive Connectivity Establishment
IETF	Internet Engineering Task Force
IM	Instant Messaging
IRC	Internet Relay Chat
ITU	International Telecommunication Union
JSEP	JavaScript Session Establishment Protocol
JSFL	JavaScript Flash Language
LAN	Local Area Networks
MCU	Multipoint Control Unit
MEGACO	Media Gateway Control Protocol
NAT	Network Address Translation
NR	Noise Reduction
QoS	Quality of Service
RFC	Request for Comments
RIA	Rich Internet Application
RTC	Real Time Communication
RTCP	Real Time Control Protocol
RTMFP	Real Time Media Flow Protocol
RTP	Real-time Transport Protocol
SCTP	System Control Transmission Protocol
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SRTP	Secure Real-time Transport Protocol
STUN	Simple Transversal Utilities for NAT
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
VoIP	Voice over IP
VVoIP	Video and Voice over IP
W3C	World Wide Web Consortium
WebRTC	Real Time Communications for the Web
WHATWG	Web HyperText Application Technology Working Group
WWW	World Wide Web
XSS	Cross-site scripting

1 Introduction

Video communication is changing rapidly, the dramatical increase of video communication between people is forcing technology to support mobile and real-time video experiences in multiple ways. The existence of reliable, low cost and simple platforms for real-time communication is becoming an essential part for the future of consumer behavior, business structure and innovation.

The world is seeing how technology and media are changing the way people interact with each other, communication over the internet is doing so by helping users to interact, talk and see content in a cheap and reliable way. This way of interacting is adding new features to users that have never been available before, they are now able to have high-engagement interaction, where richer, more intimate communication is possible. At the same time this is changing traditions and habits of communication between people and transforming personal relationships.

Distances are now shorter, bringing individuals and groups together around the world, allowing people to connect with friends and meet new people in different ways.

At the same time, it also helps businesses to have lightweight communication options that will increase their efficiency besides the size of the company. Thus, encouraging front end designers designers and device developers to turn their products into multi-functional and inter-operable communication devices.

Video has been available in the World Wide Web (WWW) since the 1990s, it has evolved to be less CPU consuming and has adapted to the new link rates while affordable digital and video cameras have become a must have feature in nowadays computers. Those two enablers along with the increased demand for richer applications for the WWW are some of the reasons behind WebRTC.

Real Time Communications for the Web (WebRTC) is a suite of protocols to enable human communications via voice and audio which this Real Time Communication (RTC) should be as natural in a web application as browsing images or visiting websites. With this simple approach WebRTC tries to transform something that has been traditionally complex and expensive to an open application that can be used by everybody, integrating this RTC technology in all existing web applications and giving the developers the ability to innovate and allow rich user interaction in their applications.

Many web services already use RTC technology to allow communication and most of them require the user to download native apps or plugins to make it work. With WebRTC video communications between users should be transparent for them since downloading, installing and using plugins can be complex and tedious for the user. On the other side, the usage of those is also complicated from the development point of view and restricts the ability of developers to come out with great features that can enrich the communication between people.

WebRTC project major guidelines are based on working Application Programming Interfaces (APIs) that are open source, free and standardized.

1.1 Background

WebRTC is an effort to bring defined APIs to JavaScript developers allowing them to code RTC applications into the web. This is commonly seen as a call system over the web or video calling applications.

Web application APIs are defined into the HyperText Markup Language (HTML) version 5 and help developers to add features to their web applications with minimal effort using JavaScript functions. APIs are protocols intended to be interfaces to communicate with other available protocols, they are used in web development to access the full potential of the browsers computing some part of the dynamic web applications on the client side. WebRTC API is being drafted by the World Wide Web Consortium (W3C) alongside with the Internet Engineering Task Force (IETF) . This API has been iterated and updated with new versions that define a better way of usage thanks to the help of web developers.

The first announcement went public in a working group of W3C in May 2011 [3] and the official mailing list started in April 2011 [4]. 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 final version. The public draft of W3C came public the 27th of October 2011 [5]. During this first W3C draft, only media (audio and video) could be sent over the network to other peers, it was focused in the way browsers are able to access the media devices without using any plugin or external software.

WebRTC project also joined the IETF with a working group in May 2011 [6]. Milestones of the IETF initially marked December 2011 as the 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 [7].

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 [8].

WebRTC APIs rely on two different coding languages, HTML and JavaScript, HTML is the de facto format for serving web applications and JavaScript is becoming the most popular scripting system for web clients to allow users to dynamically interact with the web application. The actual version of WebRTC is a merge between different API that integrate with each other to provide flexible RTC in the browser. The final goal is to allow developers to create awesome features with this protocol for those web applications that are cross compatible.

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 Web HyperText Application Technology Working Group (WHATWG) HTML specification.

1.2 Challenges

WebRTC is a suite of protocols that will share the available resources with many other applications. Due to the short experience in WebRTC congestion situations that share the available bandwidth, we will find some lack of documentation or previous literature regarding this topic compared with other existing solutions.

The aim to test and help to develop new protocols such as WebRTC is unfortunately accompanied by a lack of information that may affect some of the statements made in this thesis. Hopefully, this shouldnÕt affect its development neither its conclusions.

Considering the fact that WebRTC is being developed at the moment of writing this thesis, some of the statements made here might be different in the upcoming versions of WebRTC meaning that some of the analyzed problems could have been solved.

General WebRTC challenges are related to two different issues: technical problems and political decisions. Firstly, congestion mechanisms for RTC have always been complicated to implement due to the need of a fast response against path disturbances and link congestions. In the course of this thesis we might find limitations in WebRTC when having constrained links.

Network Address Translation (NAT) will also arise as a problem, succeeding when setting up a communication path is crucial in RTC protocols.

WebRTC is being standardized in different W3C and IETF working groups, meaning that there is a common interest to reach consensus in all aspects of this protocol. This is positive as it is supposed to adapt the protocol to all possible needs but it also delays some decisions that might affect the development of the APIs.

During the development of the thesis we will focus in the technical challenges of the protocol.

1.3 Contribution

Investigate how WebRTC performs in a real environment trying to evaluate the best way to set multiple peer connections able to transfer media 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 RTC protocols we expect to understand the way WebRTC performs when handling in different environments.

1.4 Goals

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

- WebRTC performs in different topologies and environments using multiple sources of video and audio that will be encoded with the codec provided by

the browser.

- Usage of WebRTC to build a real application that can be used by users proving that the API is ready to be deployed as well as it 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.
- Testing of different WebRTC topologies with different network constraints to observe the response of the actual existing API.

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 existing API.

1.5 Structure

Not sure about here

2 Real-time Communication

Real-time Communication can be defined as any mode of communication where users can exchange information and media with low latency, real-time aspect of it can be defined as live. The purpose of RTC is widely seen as a way to intercommunicate between people or software. This can be done in a two-way scenario where data is transmitted between both sides, being both users receivers and senders, or in a one-way configuration with one unique source of data and one or multiple receivers. In the first configuration latency is very important in order to achieve good common communication between both users whereas the second scenario can tolerate some latency in the link but data transmission must be continue. In two-way communication data can be transmitted using multiple technologies, the topologies used can be either peer-to-peer or using a centralized relay. Some other ways of transmitting data include multicast or broadcast.

In broadcast and multicast mode data is transferred to multiple peers in a network but does not require to be real time in most cases.

Figure 1: Real time communication between two users over the Internet.

Figure 1 describes an RTC scenario for two users, the technology that provide the communication may differ in each situation but the goal is always the same. RTC has two important characteristics that are always common in all technologies, there must be a signaling or agreement between the two entities, either with the central node or with the other user. This part is used by the protocols to check the capabilities of the two entities before proceeding to send the media. In this part, codec agreement and keep-alive methods are decided at the same time as all the multiple features that will be enabled in the new session, making it crucial to configure the media and data to be transmitted.

On the other hand, once signaling is done data can be sent to the receiver, this data may include media (audio or video) and data. This transmission may require also some extra signaling messages to be exchanged in order to maintain the link or adapt the constraints to the actual network conditions.

RTC can be either over the Internet or using traditional techniques, some of them are: telephony, mobile phone communication, radio, instant messaging (IM) , Voice over IP (VoIP) , Video and Voice over IP (VVoIP) , Internet Relay Chat (IRC) and videoconferencing.

All the previous ways of communication work in real time, logically they work

using Figure 1 topology but with different kind of protocols. In this thesis we will manly work with Internet RTC using media and data.

2.1 Session Initiation Protocol (SIP)

SIP allows communication between two different users with audio/video support in real-time. SIP final Request for Comments (RFC) was published in June 2004, this document describes the original functionalities and mechanisms of SIP [9]. From an overview perspective, SIP is an application-layer control protocol for multimedia sessions which can establish, maintain and terminate media sessions. 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.

This protocol work alongside with other existing technologies such as Real-time Transport Protocol (RTP) , Session Description Protocol (SDP) and Media Gateway Control Protocol (MEGACO) . Using SDP for the session negotiation between the end-points and RTP for the media transport, all these protocols are widely used in other technologies and usually provide legacy for older devices and outdated versions. 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 data.

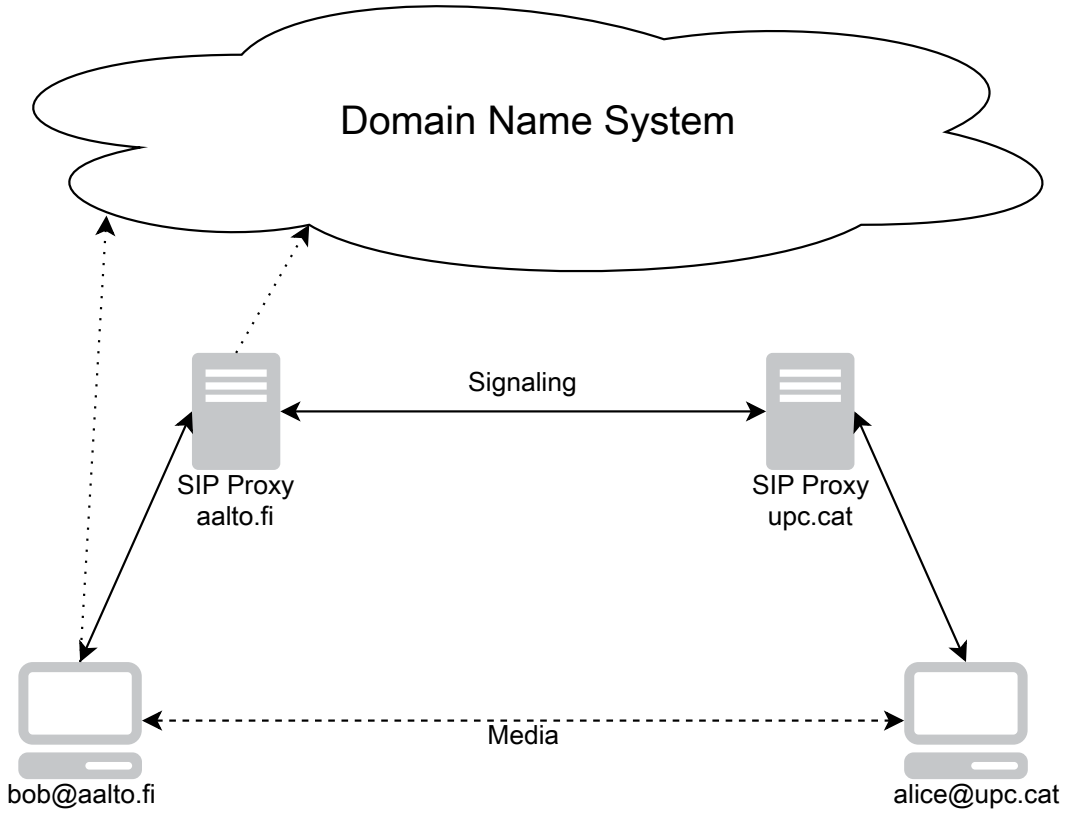


Figure 2: SIP architecture for end-to-end signaling.

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 allows the messages to be exchanged between both peers, SIP uses the following three-way handshake: INVITE, 200OK and ACK. 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 provides 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 using User Datagram Protocol (UDP) [9].

SIP is a pure VoIP confederated technology that helped the community to learn about real-time P2P communication.

2.2 Real Time Media Flow Protocol (RTMFP) and Adobe Flash

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

Adobe Flash is a multimedia software that uses a plugin to work on top of the browser, it is used to build multimedia experiences for end users such as graphics, animation, games and Rich Internet Applications (RIA) . It is widely used to stream video or audio in web applications, in order to reproduce this content we need to install Adobe Flash plugin in our computer. It also uses a different programming language that do not comply with any standards called JavaScript Flash Language (JSFL) and ActionStript. RTMFP and Adobe Flash require a plugin to work with any device, this obliges the user to install extra software that is not included in the browser, these two technologies are not standardized and are difficult to enable in some mobile devices. 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 works over UDP [10]. RTMFP provides a full suite of methods and functions that allow the browser to access the necessary mechanisms to run real-time media communication, those methods are included into the plugin that must be installed prior usage. RTMFP is a private and licensed protocol. It also handles congestion control on the packets and NAT transversal issues. One of the biggest differences is that, compared with SIP, RTMFP does not provide inter-domain connectivity and both peers must be in the same working domain to be able to communicate. This protocol is implemented by using Flash Player, Adobe

Integrated Runtime (AIR) and Adobe Media Server (AMS) [10].

Media transfer in this protocol is encrypted, this issue has been addressed clearly in RTMFP by using proprietary algorithms and different encryption 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 [11].

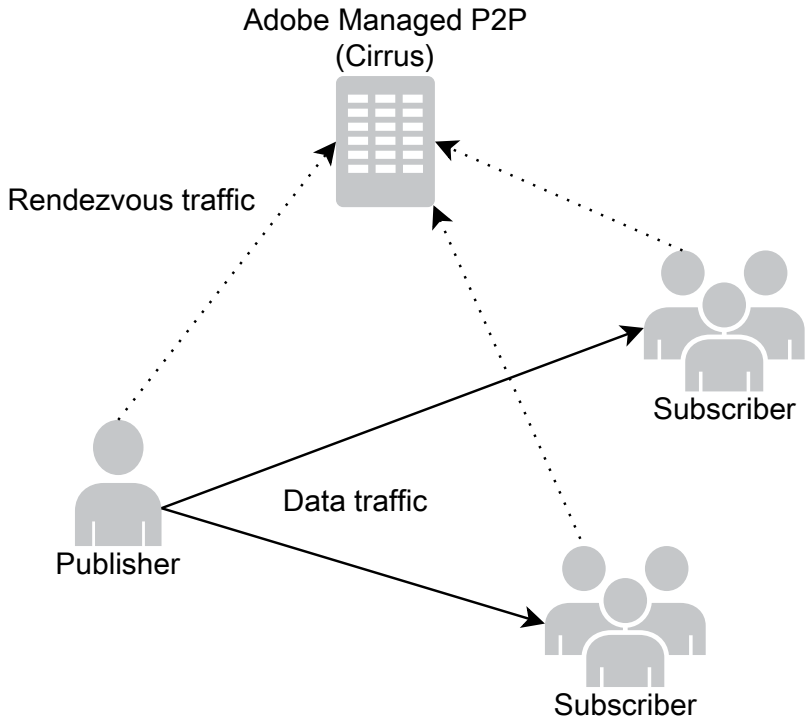


Figure 3: RTMFP architecture using Cirrus.

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.

2.3 WebRTC

WebRTC is part of the HTML5 proposal, it is defined in a W3C draft [3], and enables RTC capabilities between Internet browsers using simple JavaScript APIs. Providing video, audio and data P2P without any plugins. This API replaces the need of any plugin for P2P communications in browsers, WebRTC uses already existing standardized protocols, learned from SIP, to perform RTC.

The project was open sourced by Google to keep working with the IETF in order to standardize the technology [8].

WebRTC provides interoperability between different browser vendors, this allow the APIs to be accessible by the developers assuring high degree of compatibility

(Figure 4). Some of the major browsers that actively implement some of the WebRTC APIs are: Google Chrome, Mozilla Firefox and Opera.

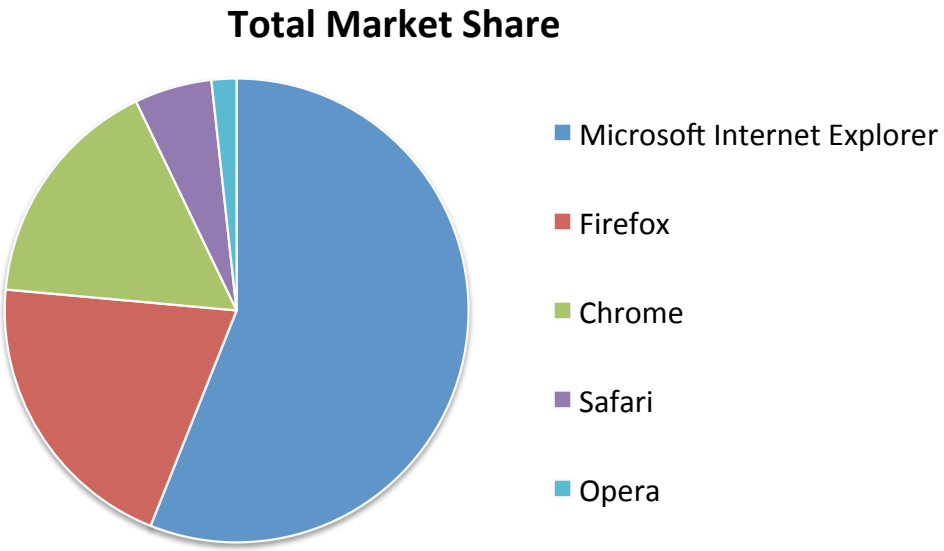


Figure 4: Market share of browser vendors by April 2013 [1].

With WebRTC developers can provide applications for nearly half of the desktop devices available, mobile devices will integrate WebRTC as part of their HTML5 package to also enable RTC soon [12].

WebRTC is composed by to important APIs that enable those features, GetUserMedia and PeerConnection. Both of them are accessible by JavaScript by the browser.

2.3.1 Device Access API

WebRTC uses an API called GetUserMedia to access media streams from local devices (video cameras and microphones). This API itself does not provide RTC, furthermore, provides media to be used as simple HTML elements in any web application. GetUserMedia allows developers to access local media devices using JavaScript code and generate media streams to be used either with the rest of the WebRTC APIs or with the HTML5 video element [2].

GetUserMedia is already interoperable between Google Chrome, Firefox and Opera [13].

This proposal was first attached directly to the WebRTC working 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.

Figure 5 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 a Document Object Model (DOM) element that reproduces local and remote media streams.

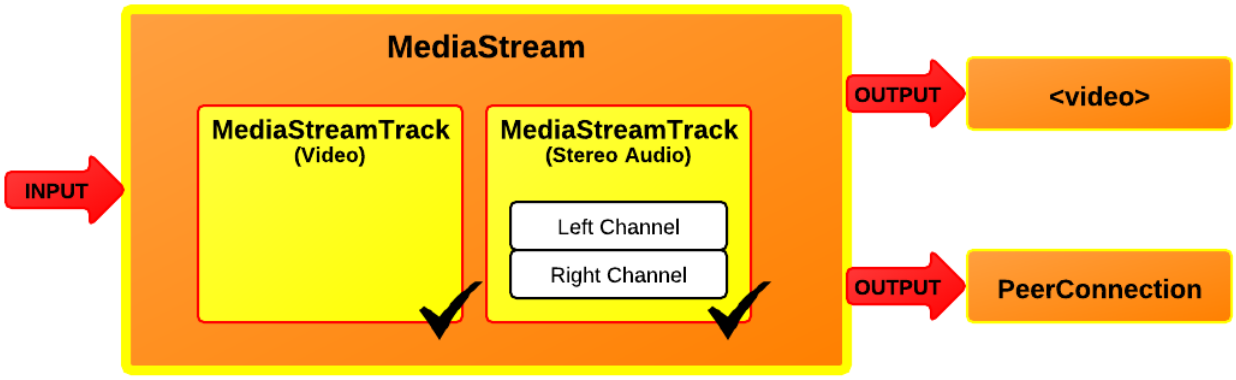


Figure 5: Media Stream API [2].

GetUserMedia API works in a fallback model, this means than the JavaScript method will return an object that can be played in an HTML web application, a simple example of this method can be seen in the following code.

Listing 1: Simple example of video and audio access using JavaScript

```
navigator.webkitGetUserMedia(cameraConstraints(), gotStream, function() {
        console.log("GetUserMedia failed");
});

function gotStream(stream) {
        console.log("GetUserMedia succeeded");
        document.getElementById("local-video").src =
            webkitURL.createObjectURL(stream);
}
```

With the previous code, we are using the video and audio media from our devices to be played in an video HTML element identified as *local-video*.

GetUserMedia also allow developers to set some specific constraints to the media acquisition. This help developers to better adapt the stream to their requirements, those *cameraConstraints()* are stored into a JavaScript Object Notation library and provided to the API through the *navigator.webkitGetUserMedia* method.

2.3.2 Networking API

WebRTC uses a separate API to provide the networking support to transfer media to the other peers, this API is named *PeerConnection* [14]. This API bundles all the internal mechanisms of the browser to enable media and data transfer, at the same time it also handles all the exchange signaling messages with specific JavaScript methods.

Signaling will be exchanged using a topology similar to Figure 6 with the signaling being sent either by WebSockets or other HTTP polling protocols. Messages are built using a modified bundled version of SDP, WebRTC is similar to SIP as it can work over RTC and existing technologies.

WebRTC uses multiplexing over one port when sending the traffic, this means that media and data are sent over the same port from peer to peer, traffic is sent by UDP or TCP [15]. This networking API provides signaling and NAT transversal techniques to bypass routers and firewalls, this part is very important to guarantee a high degree of success when establishing calls in different scenarios.

This P2P session establishment system works in a constrained environment similar to RTMFP but it has been designed to provide some degree of legacy for other SDP based technologies such as SIP. Figure 6 shows how a WebRTC simple P2P scenario works, the server used for signaling is a web server.

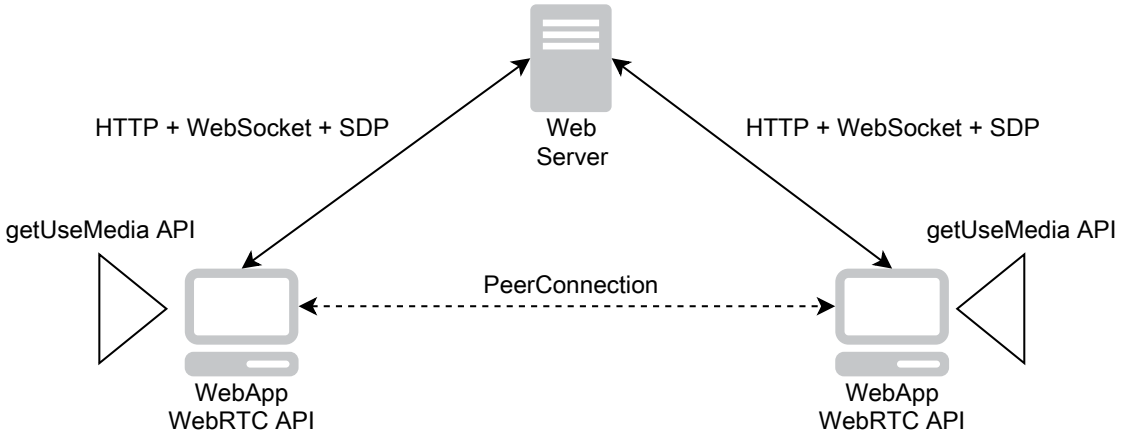


Figure 6: WebRTC simple topology for P2P communication.

Figure 6 does not show relay machines that provide NAT transversal solutions. In a real application we could host those relay machines either in the same web server or in external services, those servers must be introduced into the WebRTC *PeerConnection* configuration when starting a new call.

Listing 2: Simple example of *PeerConnection* using JavaScript

```
pc = new webkitRTCPeerConnection(servers);
pc.onicecandidate = iceCallback1;

//Localstream is the local media obtained with the GetUserMedia API
pc.addStream(localstream);

function iceCallback1(event){
        if (event.candidate) {
                sendMessage(event.candidate);
        }
}

//When incoming candidate from the other peer we send it to the
    PeerConnection
pc.addIceCandidate(new RTCIceCandidate(event.candidate));
```

```
//This is fired when the remote media is received
pc.onaddstream = gotRemoteStream;
function gotRemoteStream(e){
        document.getElementById("remote-video").src =
            URL.createObjectURL(e.stream);
}
```

Previous code describes a simple example on how to use the *PeerConnection* API to perform a P2P connection and start transferring media, this code works in conjunction with the code in section 2.3.1. When building the new *PeerConnection* object we need to pass the JSON object *server* with the relay configuration for the NAT transversal process.

2.3.3 Control and Monitoring API

Control and monitoring is an important part of all RTC protocols, this part is usually handled by the software that adapts the constraints and configurations to the available resources.

In WebRTC, this is done through the Statistics Model and Constraints defined in the W3C draft [14], these methods are part of the actual *ÊPeerConnection* API defined in section 2.3.2. Once the *PeerConnection*Êis made and media is flowing we need to measure the quality of the connection, this is done by retrieving the stats provided in the Real Time Control Protocol (RTCP) messages that are being sent over the link.

To access this data contained on the control messages we need to call the *getStats()* method in the *PeerConnection*, this method will allow the developers to access that data in a JSON format that will require some post-processing. Statistical models will be useful for the developers to monitor the usage of their WebRTC applications and change the attributes of the *PeerConnection*.

With constraints developers are able to change media capture configuration by setting Frames per Second (FPS) and video resolution. Other attributes can be set on the *PeerConnection* such as bandwidth requirements, transfer rate is automatically adjusted in WebRTC using its internal mechanisms but we can set a maximum value.

JSON objects for camera and bandwidth constraints must be defined as in the following code.

Listing 3: JSON objects for constraints attributes in WebRTC

```
//Media constraints
var constraints = {
        "audio": true,
        "video": {
                "mandatory": {
                        "minWidth": "300",
                        "maxWidth": "640",
                        "minHeight": "200",
```

```
                    "maxHeight": "480",
                    "minFrameRate": "30"
                },
        "optional": []
        }
}

//Bandwidth
var pc_constraints = {
        "mandatory": {},
        "optional": [
         {
                "bandwidth": "1000"
        }
        ]
}
```

Both constraints objects are added to the *GetUserMedia* and *PeerConnection* methods when building the new object. Values are in pixels for the media attributes and Kbit/s for the rate configuration.

2.3.4 Low vs High level API

During the development of WebRTC there has been a lot of discussion in the different working groups about the API layout, those APIs have been designed using the feedback provided by the developers and experts on the area.

One of the difficult parts in the standardization process has been to decide the complexity level of the API, how much is available to be accessed by the developers and which configurations or mechanisms should be automatized in the browser. After long discussion, WebRTC is now working with JavaScript Session Establishment Protocol (JSEP) [16], this API is a low level API that gives the developers control of the signaling plane allowing each application to be used in different environments, some will give legacy to SIP or Jingle protocols meanwhile others might only work in a closed web domain.

The media process is done in the browser but most of the signaling is handled in the JavaScript plane by using JSEP methods and functions. Figure 7 represents the JSEP signaling model, this system extracts the signaling part leaving media transmission to the browser. However, JSEP provides mechanisms to create offers and answers, as well to apply them to a session. The way those messages are communicated to the remote side is left entirely up to the application.

One interesting feature that JSEP provides is called *rehydration*, this process is used whenever a page that contains an existing WebRTC session is reloaded keeping the existing session alive. This will help to avoid session cuts when accidentally reloading the page or with any automatic update from the web application. With *rehydration* the current signaling state is stored somewhere outside the page, either on the server or in browser local storage [16].

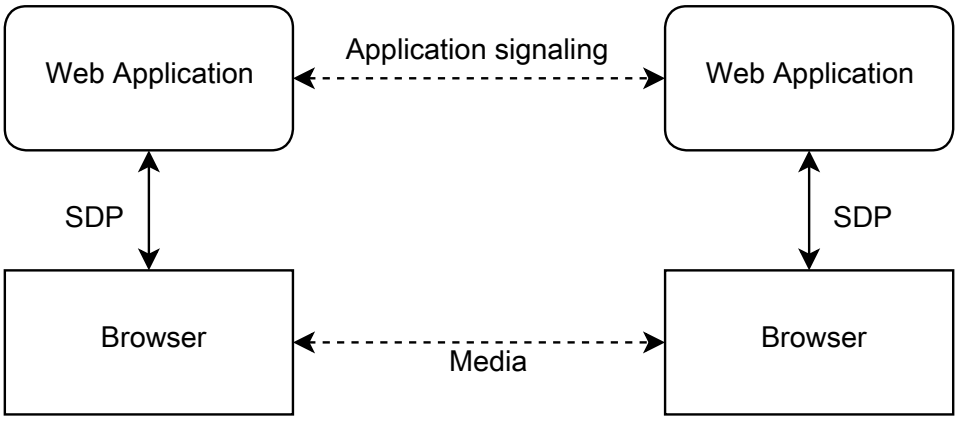


Figure 7: JSEP signaling model.

Low level APIs allow developers to build their own high level APIs that handle all the WebRTC protocol from media access to signaling. Those high level methods are useful to simplify the way JavaScript developers use their applications, building object oriented calls we can have JavaScript libraries that set up and maintain multiple calls at the same time. The benefits of having low level JSEP API for WebRTC are the multiple possibilities to adapt WebRTC to the requirements of each specific application.

In this thesis we will be using some high level APIs that handle signaling over WebSockets and statistics.

2.3.5 Internals of WebRTC

WebRTC has multiple internal mechanisms to enable the RTC on the browser level by APIs. Those mechanisms work together to accomplish all the needs for WebRTC features, some of them are related to the network level or video acquisition.

One of WebRTC main issues is NAT transversal, this problem will affect all RTC technologies. Real-time P2P protocols cannot success in all environments without a NAT transversal solution, in SIP and WebRTC this problem is addressed with the usage of Simple Transversal Utilities for NAT (STUN) and Traversal Using Relays around NAT (TURN) [17] [18]. Both of these methods are combined with Interactive Connectivity Establishment (ICE) technique that helps WebRTC to decide which is the best way to bypass NATs and firewalls, ICE is widely used in P2P media communications and has proven to be reliable when choosing the best option to succeed with the connexion [19].

TURN and STUN machines are usually placed outside the local network of the clients and help them to find the way to communicate each other by discovering new open paths, the final decision is taken by the ICE mechanism in WebRTC. STUN server provide different IP and port configurations that allow a direct connection to the peer behind the firewall, those configurations are named *candidates*, this information is given to the sender that process the information and tries to choose the best *candidate*. On the other side, TURN works as a relay, this option should

be always stated as the last resort when there is no valid *candidate* for connectivity. TURNs will reroute there traffic from one peer tot the other.

All traffic in WebRTC is done over UDP except the case of TURN TCP and multiplexed over the same port.

Media encoding in WebRTC is done through codecs implemented in the browser internals, those codecs are decided in the IETF working group and have been discussed for long time.

Defined codecs for audio are G711 and Opus. G711 is an International Telecommunication Union (ITU) standard audio codec that has been used in multiple real time applications such as SIP. In real-time media applications Opus is also a good alternative for G711, Opus is a lossy audio compression format codec developed by the IETF and that is designed to work in real-time media applications on the Internet [20]. Opus can be easily adjusted for high and low encoding rates being a good candidate for the needs of WebRTC.

Along with the codecs, the audio engine for WebRTC also includes some interesting mechanisms such as Acoustic Echo Canceler (AEC) and Noise Reduction (NR) . The first mechanism is a software based signal processing component that removes, in real time, the acoustic echo resulting from the voice being played out coming into the microphone, with this, WebRTC avoids to create audio loops with the output and input sound devices of computers. NR is a component that removes background noise associated to real time audio communications.

Those mechanisms provide a smooth audio input for WebRTC protocol.

There has been a lot of discussion regarding video codec, two of the proposed codecs are H.264 and VP8. H.264 is a standard codec for video compression, this codec is widely used for recording and transmission of high definition video. Originally it was also selected due its high compatibility with existing devices and software, H.264 has made some controversy as it is patented and licensed by MPEG LA and may add some patenting problem for WebRTC. VP8 is a video compression codec owned by Google released in May 2010, VP8 is supported by Chrome, Opera and Firefox by default and is the de facto codec for WebRTC by May 2013. Later on, Google announced a VP8 patent cross-license agreement to provide royalty-free license to allow developers to implement VP8 video in their web applications [21]. This video codec is adaptive and performs well in low bandwidth links at the same time as providing royalty-free implementation.

WebRTC is not only useful for sending media, it can also provide P2P data transfer. This feature is named *Data Channel* and provides real time data transfer, this can be used with multiple purposes, from real time IM service to gaming, but it is interesting to as *Data Channel* allows generic data exchange in a bidirectional way between two peers [22]. Non-media data type in WebRTC is handled by using System Control Transmission Protocol (SCTP) encapsulated over Datagram Transport Layer Security (DTLS) [23] [24] [22].

The encapsulation of SCTP over DTLS on top of ICE/UDP provides a NAT traversal solution that combines confidentiality, source authentication and integrity with protected transfers. This data transport service can operate in parallel with media transfer and is sent multiplexed over the same port. This feature of WebRTC

is accessible from the JavaScript *PeerConnection* API by a combination of methods, functions and callbacks. From the developer perspective all the previous statements regarding security and transport are made in the browser internals providing a simple and reliable way of sending P2P secure data over WebRTC.

WebRTC provides Secure Real-time Transport Protocol (SRTP) to allow media to be secured.

Quality of Service (QoS) for WebRTC is also being discussed in the IETF and a draft is available with some proposals [25]. 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 [25].

WebRTC also uses a Google congestion control algorithm that enables proper congestion control mechanisms for rate adaptation [26]. The aim of this algorithm is to provide performance and bandwidth sharing with other ongoing conferences and applications that share the same link.

2.3.6 Security concerns

To handle the signaling in WebRTC we use a web server, this web server will exchange the message between the peers in multiple different ways. Even this system provides high flexibility for developers to allow multiple scenarios, it also has some important security concerns [27]. Figure 6 represents the simple topology for a WebRTC call, web server relays the signaling messages to the peers and the media transport is done between them and handled by the browser.

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 can also become a threat.

Actual browsers execute JavaScript scripts provided by the accessed web sites, this also includes malicious scripts, but in the case of WebRTC this points out some privacy problems. 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 [28]. 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.

In web services, issues such as Cross-site scripting (XSS) provide high risk of privacy vulnerability. Those situations, shown in Figure 8 are given when a third-party server provides JavaScript scripts to a different domain, this script cannot be trusted by the original domain that the user is accessing and could trigger browser actions

that could harm the privacy. For example, in WebRTC, we could load a malicious script from a third-party entity that builds a WebRTC call to an undesired receiver without the user noticing this problem. Nowadays, browsers provide some degree of protection against XSS and do not let some scripting actions to be performed.

Other related vulnerabilities in WebRTC APIs is the possibility to establish media forwarding to a third peer, for example, once the user has accepted the access to the media the provided JavaScript will build one *PeerConnection* to the receiver and one to a remote server that could store the call without the user noticing. Those problems are not only related to WebRTC and could be given in related protocols.

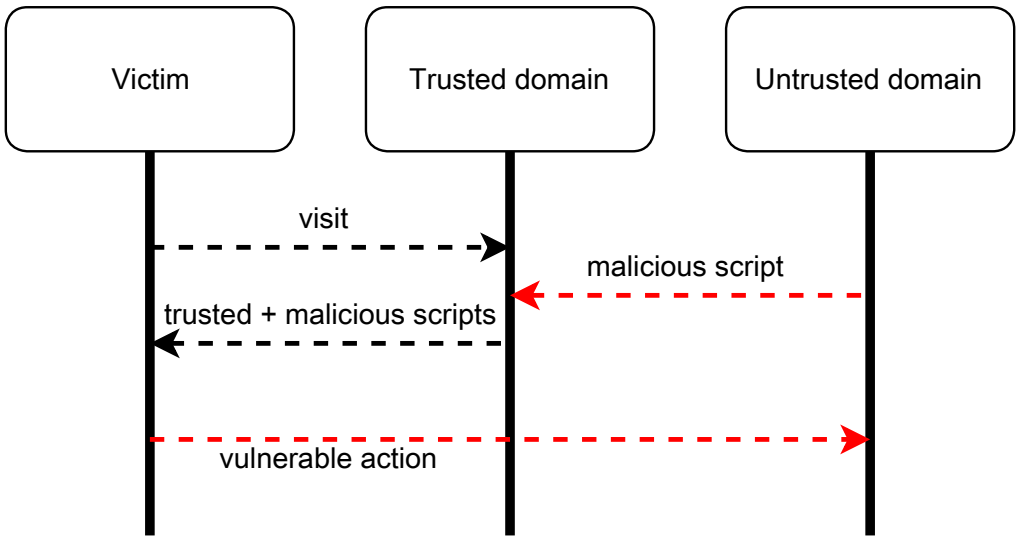


Figure 8: Example of cross-site scripting attack.

WebRTC calling procedure is done 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 Hypertext Transfer Protocol Secure (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 using some Identity Provider (IdP) that supports OpenID or BrowserID to demonstrate their identity [29]. Usually this problem is not particularly important in a closed domain, cases where both peers are in the same social network and provide 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 9.

If the web service is running over a trusted HTTPS certificate and has been authorized access to the media will be automatically after the first time, otherwise, the user will have to verify the access each time. 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 working groups.

Security an privacy issues in WebRTC can be given in multiple layers of the protocol, the increment of trust for the provider gives some vulnerability issues that sometimes cannot be easily solved if the aim is to keep a flexible and open

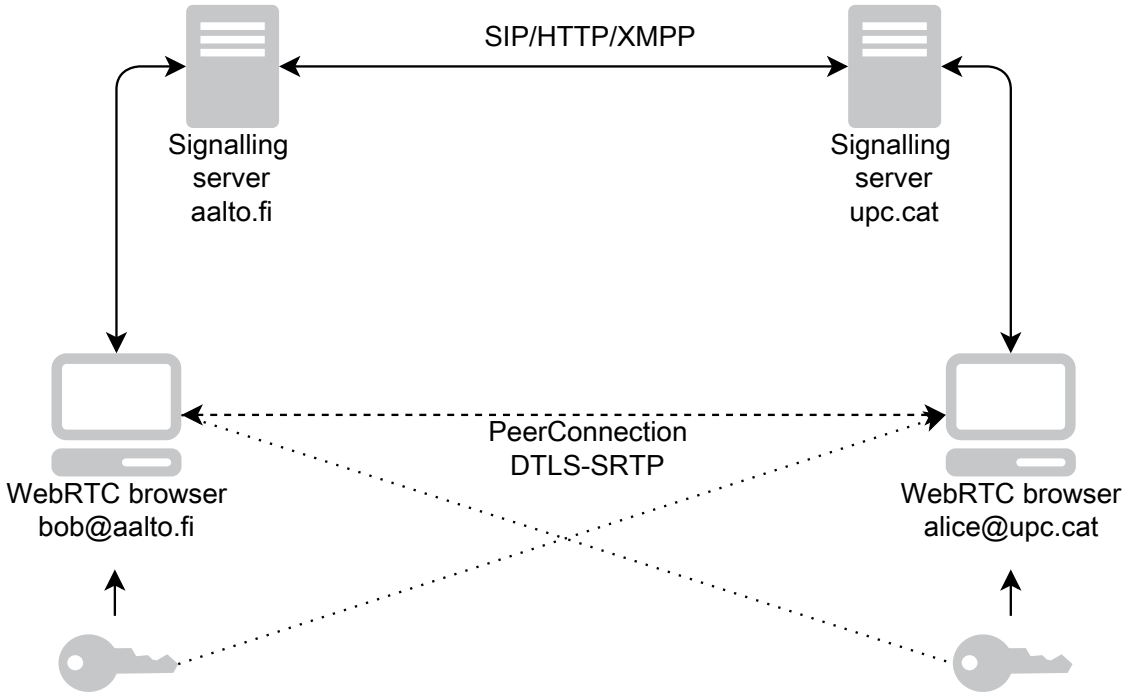


Figure 9: WebRTC cross-domain call with Identity Provider authentication.

sourced real time protocol. Some use cases for WebRTC also incorporate some level of vulnerability as the JavaScript will be provided by a third-party and could led to privacy vulnerability, in the use case of media streaming, advertisement or call centers those providers could pick data form the users and store them for further usage [27].

2.4 Comparison of WebRTC, SIP and RTMFP

After describing various RTC mechanisms and all the similar alternatives for WebRTC, Table 1 is a summary of common features between SIP, RTMFP and WebRTC. In this Table 1, common internal mechanisms are described for all of them.

RTFMP is a proprietary protocol which mean that might have its own mechanisms other than the standardized ones stated on the table to solve some of the issues.

All there protocols are designed to provide the same real time functionalities but in different ways, meanwhile SIP is a protocol that helped to develop some of the important mechanisms that will be used in other technologies is still not easily accessible by developers. On the other side, RTMFP provides real time communication for developers in a licensed way having some of their mechanisms not standardized and with compatibility issues.

From the mobile perspective, SIP is used in mobile technology and WebRTC has announced to be compatible with near versions of iOS and Android [12]. Furthermore, RTMFP has active support for Android but is still not able to extend its usage to iOS platforms.

All three protocols provide NAT traversal solutions but RTMFP is the only one that provides a proprietary solution that is not standardized, SIP and WebRTC use a conjunction of TURN, STUN and ICE mechanisms.

	SIP	RTMFP	WebRTC
Plugin-enabled	No	Yes	No
Cross-domain	Yes	No	No
Licensed	No	Yes	No
Mobile	Yes	Partially	Yes
Audio	Yes	Yes	Yes
Video	Yes	Yes	Yes
Data	No	No	Yes
TURN	Yes	No	Yes
STUN	Yes	No	Yes
SDP	Yes	No	Yes
RTP	Yes	No	Yes
SRTP	Yes	No	Yes
UDP	Yes	Yes	Yes
TCP	No	No	Yes
SCTP	No	No	Yes
VP8	No	No	Yes
H.264	Yes	No	Yes
G711	Yes	No	Yes
Opus	No	No	Yes

Table 1: Feature comparison between SIP, RTMFP and WebRTC.

All of them are valid options, in this thesis we will work with WebRTC and its related mechanisms,

3 Topologies of real-time multimedia communication

This chapter will discuss different possible topologies that can be used along with WebRTC.

A topology can be defined as the arrangement of the various nodes of a network together, those nodes can be connected through different links and forms. Those topologies may have different logics and paths to have optimal performance for each specific technology. In WebRTC, we want to study how they perform in the most common use cases for real time multimedia communication.

Some challenges will be common in all the topologies described in this chapter. For example, NAT traversal problems will decide either if the call is established or not, this problem will be solved with the usage of TURN and STUN, but in some restrictive environments it might be impossible to succeed with the call establishment.

For some topologies that include the establishment of multiple *PeerConnections* the resource consumption can be a big problem. Considering that this relies in how the operating system architecture handles processes, the CPU and memory usage of WebRTC might be seen as a constraint for those topologies. For example, in Unix based systems every tab of a browser is treated as a separate process meanwhile in Dos systems this might be different. Media encoding will consume most of those resources being a bottleneck for some scenarios.

3.1 Point-to-Point

The simplest possible topology is a permanent constant link between two peers, this model is widely used in telephony and provides reliable real time communication between users. In WebRTC, point-to-point topologies work only within people in the same domain opposite to many telephony alternatives.

With point-to-point topology we can have traditional dedicated paths where the resources are reserved for each call. In Local Area Networks (LAN) we can have dedicated paths between two WebRTC users, this path can go through the switch or relay but it is unlikely that will change the routing. For WebRTC calls over the public internet the link will likely change at any time trying to get the best option path with less congestion, this is done in packet-switching technologies where the path is set up dynamically.

From the use cases perspective, multiple environments will require point-to-point topology, direct calls between two users or real time communication for IM can be possible scenarios.

Specific uses can be communication between doctor and patient in a medical web application that is cross-platform compatible and uses an WebRTC. Communication in other cases such as citizens and authorities could also rely in a WebRTC application.

3.2 One-to-Many

One-to-many or star topologies are one of the most common network topologies for media streaming, this kind of topology consist of a simple central node that transmits streams to the rest of nodes connected to it. In Figure 10, the central node might be also receiving real time data in difference of the traditional streaming scenarios and this could provide feedback communication between the receivers and the central node.

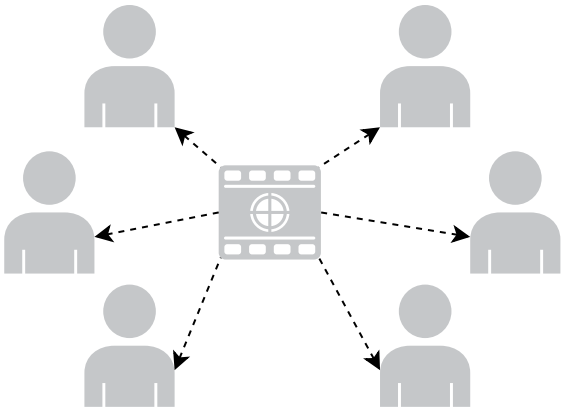


Figure 10: One-to-many topology for real time media.

Star 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.

Some of the problems are the high dependency of the central node, in case of failure the node video streaming will stop and all the connectivity with the peers will be over. On the other side, this topology is also good as it provides reliability in case of failure of one of the connected nodes as the rest of the network won't notice any difference on the communication.

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. All the previous examples will take advantage of WebRTC by having direct feedback from the connected nodes, actual media streaming technologies do not provide this kind of communication between the viewer and the origin.

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 *PeerConnection* running, central node performance will be a big constraint in this topology.

Observing other topologies, in this case, latency on the network is not as important as the rest due to the one-way communication only, in some scenarios video and audio is not required to be received on the source so 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 and data.

3.3 Many-to-Many

Many-to-many topologies are also known as mesh, this style of topology is used in multiple VoIP systems for conferencing purposes. Conferences are used in enterprises for long-distance communication between employees and working groups, by this, the need of having those calls working with good response for all participants is very important.

In a full mesh topology all peers connect between them growing the number of connections and used resources.

The value of fully meshed networks rely on the number of subscribers, the amount of *PeerConnections* stablished in a mesh network will depend on the amount of people in the conference. The number of *PeerConnections* will grow using Equation 1.

$$c = \frac{n(n-1)}{2}$$

c : Number of *PeerConnections*
n : Nodes in the mesh (1)

Equation 1 calculates the amount of WebRTC connections required for a *full mesh* topology.

3.4 Multipoint Control Unit (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.

In real time media topologies MCU is a common component, used as relay it helps end devices to handle less load multiplexing all the streams of the call into the same channel, we can have multiple peers connected to the same MCU machine that will multiplex the media sent by them into one stream forwarded to all the participants.

MCUs have to encode and decode media on the fly, this is also difficult in real time applications but can provide different encoding options to adapt the output to the link conditions.

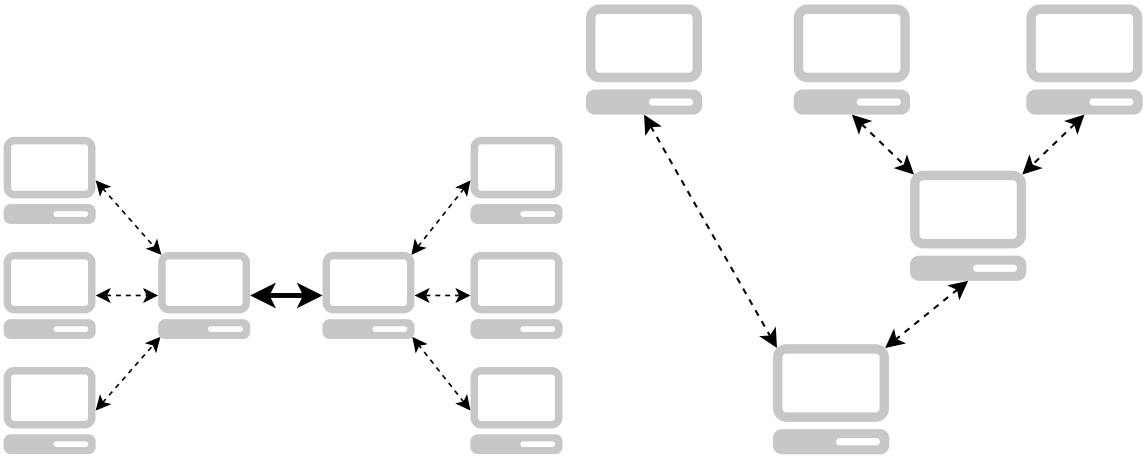
Drawbacks of MCU model rely on the dependency of the end nodes from the MCU, if the MCU fails to give good latency and performance the call quality will be affected and receivers won't get the expected response. Load in the MCU can be very high when multiple conferences are being stablished, this will require abundant resources and good throughput.

3.5 Overlay

Topologies that contain overlay techniques are those that require the media to be forwarded from one peer to the other, this kind of behavior is given in multiple peer topologies such as *hub and spoke* or *tree* seen in Figure 11.

Generally, in multiple peer scenarios we can combine all of the following structures to build a topology that fit our requirements.

WebRTC does not provide native support for overlay topologies but it is planned to implement those features in future versions of the API. Traditionally overlay has been used for media streaming over the internet.



(a) Hub and spoke topology (b) Tree topology

Figure 11: Overlay topologies.

3.5.1 Hub and spoke

Spoke-hub distribution is a topology composed by nodes and arranged like a chariot wheel. Traffic moves along spokes that are connected to the hub at the center. This type of topology, represented in Figure 11a, is good as it requires less connections to perform a full communication in the network.

This is a centralized model, by this we might have problems if the key nodes of the topology are down. It also relies in one or multiple trunk paths that can be crucial for the success of the streaming, those paths should provide good throughput and low delay.

In some technologies that rely in hub and spoke the central nodes are usually picked from the end users, calculating the best response from the users the topology is able to select the optimal candidate where the rest of nodes will connect to. When this happens that node is handling and forwarding more data that in a standalone call.

This topology uses the concept of overlay previously described.

Hub and spoke concept is also used in world logistics to distribute products and goods around the globe, focusing in bridges over the continents goods in Europe

are distributed within our internal network and shipped to other continents from a centralized node.

3.5.2 Tree

Tree topology is based on a node hierarchy, the highest level of the tree consist of a single node that is connected to one or more nodes that forward the traffic to the other layers of the topology. Tree topologies are not constrained by the amount of levels and can adapt to the required amount of end users as seen in Figure 11b.

This type of topologies are scalable and manageable. In case of failure it is relatively easy to identify the broken branch of the tree and repair that node.

On the other side, we will have connectivity problems if a node fails to keep the link up, all the layers under that node will be affected and the media forwarding will stop.

Overlay is crucial for this topology that is widely used in media streaming, for real time communications this topology won't be the best candidate due to the delay when forwarding the packets.

Topologies such as tree are not only used for media streaming but they can also be used to provide wireless coverage in difficult areas, acting as hotspots each hop can extend the coverage of the wireless in mobile applications.

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.

The second option is called congestion error meanwhile the loss of packets by buffered queues or other issues is called bit-error, over use of a link may led to losses due to congestion. In WebRTC we are using RTCP protocol for control and reporting of the ongoing stream [30].

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.

$$\frac{PKT_{loss}(T) - PKT_{loss}(T-1)}{PKT_{received}(T) - PKT_{received}(T-1) + PKT_{loss}(T) - PKT_{loss}(T-1)} \tag{2}$$

Equation 2 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) and One-way delay (OWD)

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.

OWD and RTT measurements are included in standard RTCP specification, in order to calculate this timestamp from sender and receiver is needed in the reports. Sender Report (ST) timestamp is saved in the sender meanwhile the receiver informs the same timestamp in the Receiver Report (RR) that goes back to the origin. By that, we are able to calculate the RTT using the following Equation 3.

$$RTT = TS_{RR} - TS_{SR} - T_{Delay}$$

TS_{RR}: Local timestamp at reception of last Receiver Report

TS_{SR}: Last Sender Report timestamp

T_{Delay}: Receiver time period between SR reception and RR sending in the sender (3)

Calculating OWD requires both machines clock to be accurately synchronized, we might try to assure this but usually OWD delay is defined as $\frac{RTT}{2}$.

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.

Furthermore, throughput can be divided into sending rate (BR_S), receiver rate (BR_R) and goodput (GP). From the technical point of view sender rate is defined

as the amount of packets that are injected into the network by the sender, receiver rate is the speed at which packets arrive at the receiver and goodput is calculated by discarding all the lost packets on the path, only packets that have been received are counted making goodput a good metric for our purposes. Usually, those metrics are calculated by extracting the information from the RTCP packets, in our case, we will rely on the Stats API that uses the WebRTC API to obtain the amount of bytes and measurements to manually calculate those by using JavaScript. Taking into account the last amount of bytes received, the actual amount and the time elapsed we will be able to calculate and accurate value for the goodput or throughput.

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.

4.5 Summary of metrics

Performance metrics for WebRTC will help us to determine the behavior of the link just by using information provided by the RTCP packets and the pure API, the goal of all this metrics is to provide better mechanisms to properly adapt the rate and response to the condition of the link. Rate adaptation will be a key mechanism to provide good response in WebRTC and we will closely study how those perform in

different environments, some such as RTT, OWD and setup call time will affect the user experience more than the pure video quality of the call and should also be taken in consideration. Based on the results we get for those indicators we will determine wether the congestion mechanisms are worked as defined or should be improved in the actual version of WebRTC.

5 Evaluation Environment

We have set up a testing environment to help us run tests for WebRTC. Figure 12 describes the functional blocks used for the simple video call.

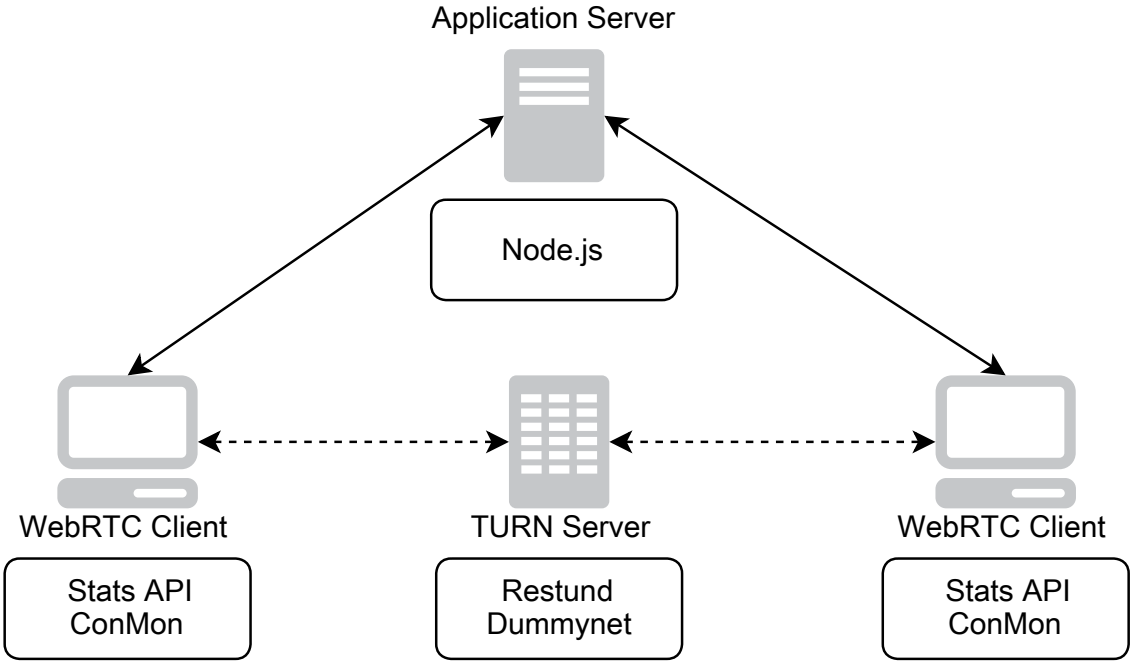


Figure 12: Description of testing environment topology.

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 [31]. 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 13 represents one video stream from the same call as Figure 14 but captured from the *ConMon* application.

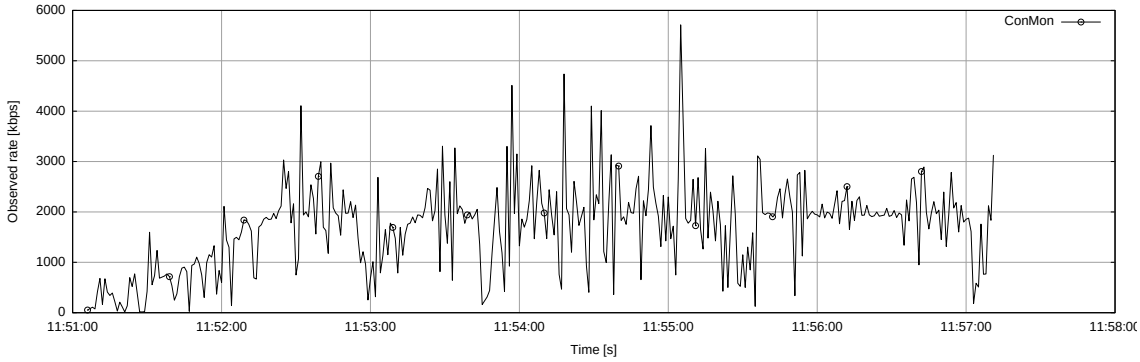


Figure 13: Point-to-point WebRTC video stream throughput graph using ConMon over public WiFi.

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.

Furthermore, *ConMon* will be used to provide OWD and RTT calculations for our tests, in order to do this we must assure a proper synchronization between local clocks in all the peers. This will be done by using the sequence number of all RTP packets captured and subtracting the timestamp saved from both sides, no RTCP will be used in this case.

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 [32].

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 [14].

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 14 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*.

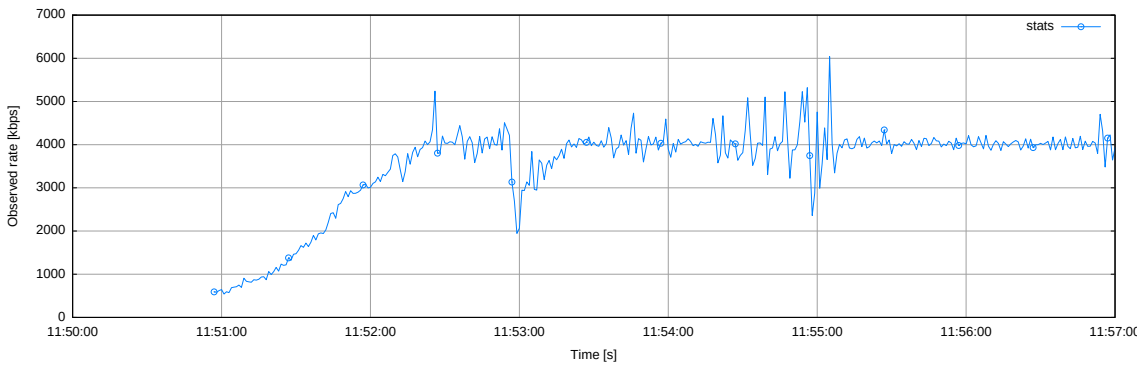


Figure 14: Point-to-point WebRTC video call total throughput graph using Stats API over public WiFi.

The previous Figure 14 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 15 and 16 plot two video streams being captured from Stats API and *ConMon*.

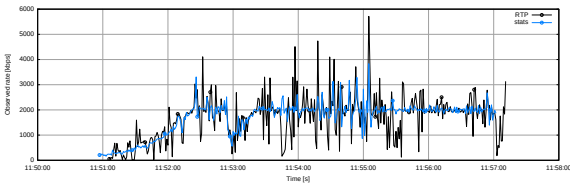


Figure 15: P2P incoming video stream comparison between ConMon and Stats API over public WiFi.

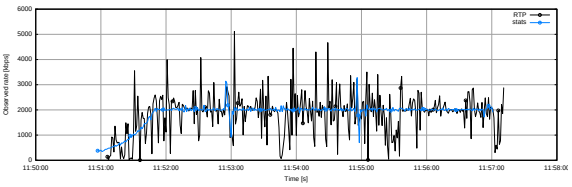


Figure 16: P2P outgoing video stream comparison between ConMon and Stats API over public WiFi.

Figure 15 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 16 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 17 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 18 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

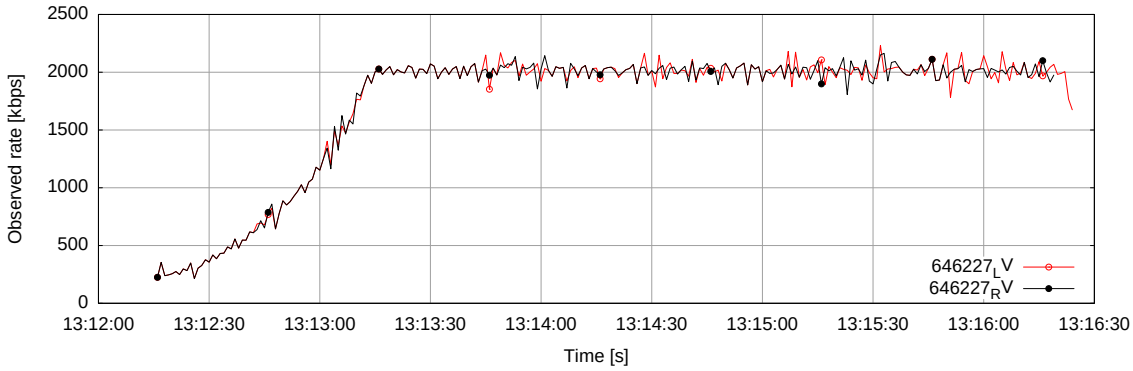


Figure 17: Video stream bandwidth using webcam input.

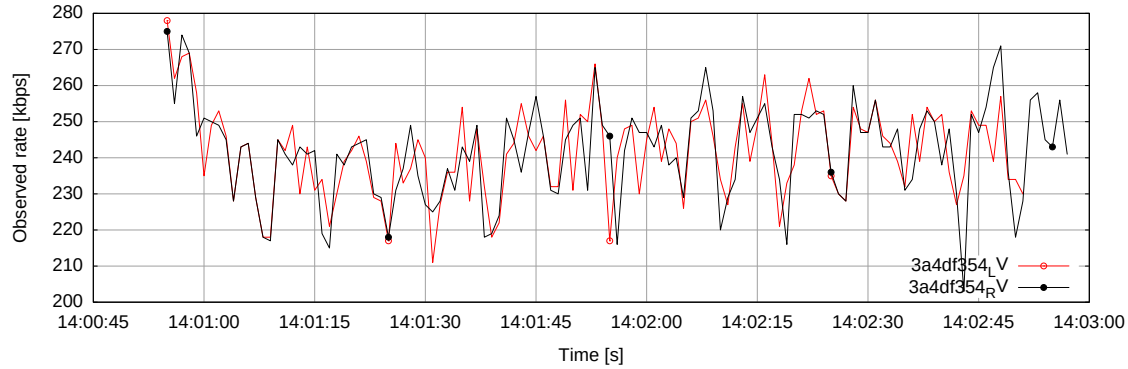


Figure 18: Video stream bandwidth using Chrome default fake video input.

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.

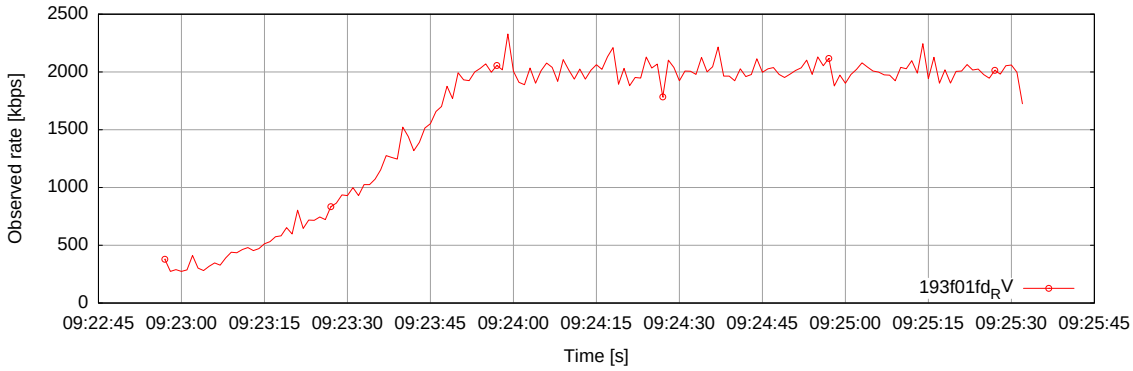


Figure 19: Video stream bandwidth using V4L2Loopback fake YUV file.

Figure 19 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 [33]. 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.

Listing 4: Forcing a hardcoded password in our TURN server

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

Furthermore, in order to force WebRTC to use TURN connectivity instead of STUN or direct path we need to configure the WebRTC API with this server by doing:

Listing 5: Configuring our TURN server in WebRTC

```
var pc_config = {
       "iceServers": [{url: "turn:XXX@192.168.1.106:3478",
           credential:"YYY"}]
};
```

The IP address will be pointing to our TURN server with the desired port (3478 by default), now all candidates are obtained through our TURN. This does not mean that the connection will run through the relay as WebRTC will try to find the best path which is still not through the TURN, to force this we will need to drop all candidates that do not point to the relay.

Listing 6: Dropping all candidates except relay

```
function onIceCandidate(event) {
       if ((event.candidate) &&
           (event.candidate.candidate.toLowerCase().indexOf('relay')) !==
           -1) {
               sendMessage({
                       type: 'candidate',
                       label: event.candidate.sdpMLineIndex,
                       id: event.candidate.sdpMid,
                       candidate: event.candidate.candidate
```

```
                },receiver,from);
        } else {
                console.log("End of candidates.");
        }
}
```

Function *onIceCandidate* will be fired every time we get a new candidate form our STUN/TURN or WebRTC API, those candidates need to be forwarded to the other peer by using our method *sendMessage* through *WebSockets*. In this code we are dropping all candidates except the ones containing the option *relay* on it, those are the candidates that will force the Peer Connection to go through our TURN machine.

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 [34].

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 in different rooms, 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 uses WebSocket protocol to carry the signaling messages from peer to peer.

5.5 Summary of tools

Using all the previous mentioned tools together we will be able to measure how WebRTC performs in a real environment, some tools have been modified according to our requirements of bandwidth and security. To process the data obtained by all those tools we have built some special scripts that measure and extract the information we require form the captures, some of them are explained in Appendix C.

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.

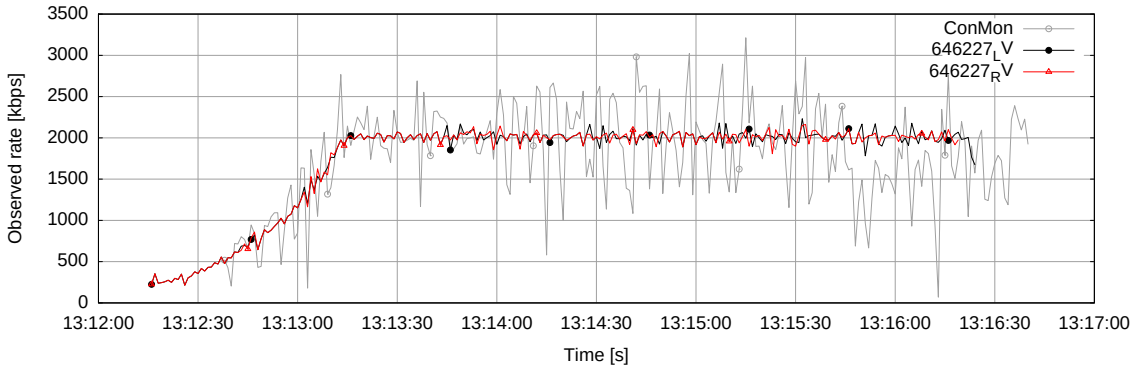


Figure 20: Point-to-point video stream plot using StatsAPI and ConMon data over WiFi.

Figure 20 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.

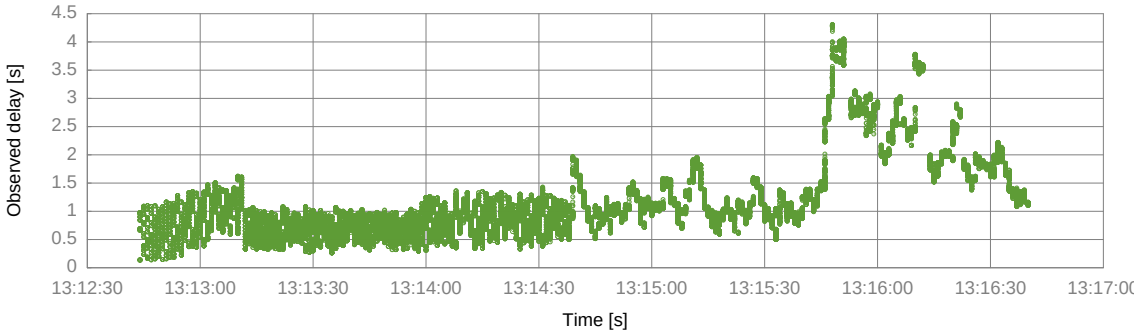


Figure 21: Delay calculated on the same stream captured using ConMon in both ends over WiFi.

Figure 21 represents the delay of the stream plotted in 20. We can see that the quality of the call is affected by the network distortion at the end of Figure 20, 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.

6.1.2 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 22 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

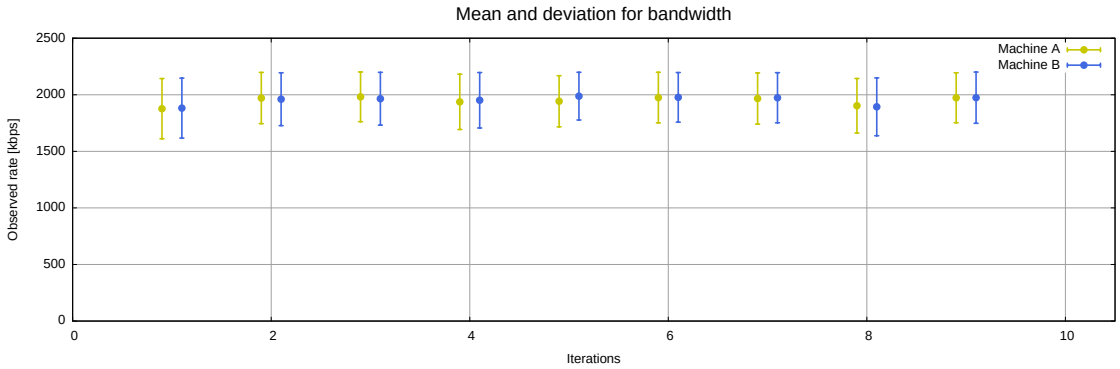


Figure 22: Bandwidth results for non-conditioned link.

233 Kbit/s of deviation which gives the conclusion of having approximately 1 Mbit/s 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 2. 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. We also had zero packet losses and two calls that failed to succeed using TURN in the standard environment.

	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
Bandwidth (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

Table 2: P2P test with no link conditions.

Delay values in Table 2 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 21 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.

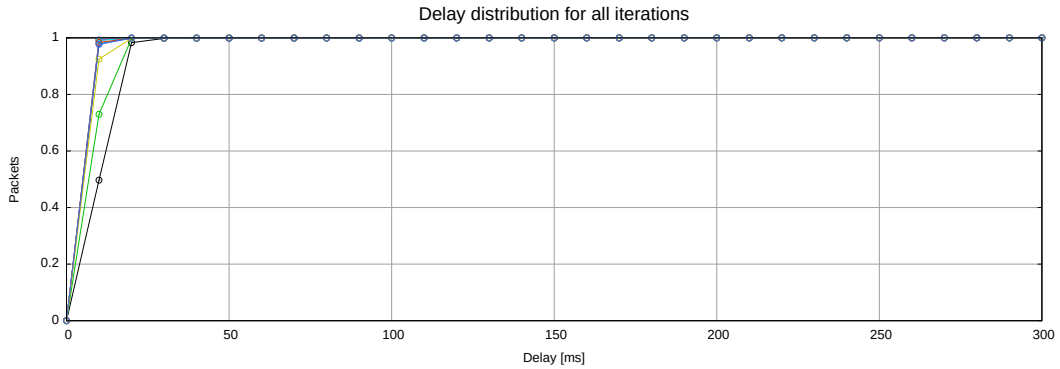


Figure 23: Delay distribution in each P2P iterations with no link constraints.

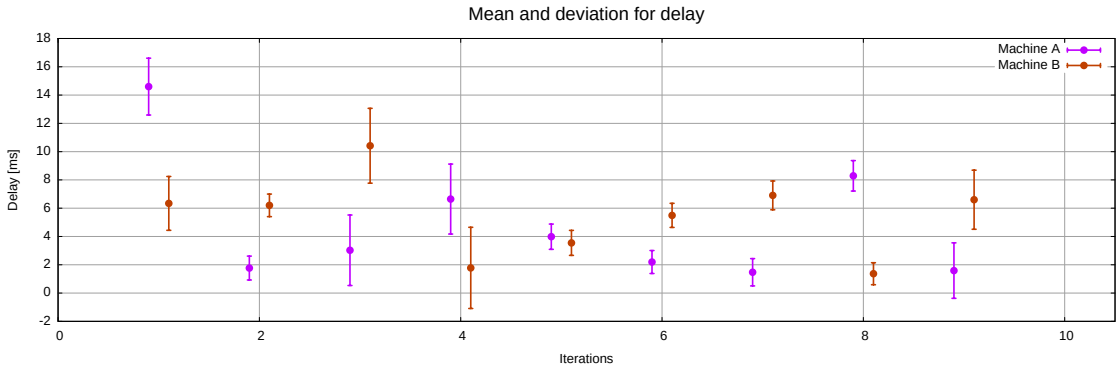


Figure 24: Mean and deviation for delay in each P2P iterations with no link constraints.

Figure 24 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 23, 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 24 and 23 differentiate from Figure 21 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 21 is good to observe just one call if we add some conditions to the link meanwhile the call is going on.

6.1.3 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 3 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%.

	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 3: Averaged bandwidth with different packet loss conditions.

The amount of packets lost in every test is slightly lower than the exact percentage of loss because the use of Forward Error Correction in WebRTC in Chrome, this mechanism is used to control errors in data connection with noisy channels that led to packet losses. FEC is not a must feature to implement in WebRTC but Chrome carries it as default.

When using FEC the sender encodes the message in a redundant way, by having this redundancy the receiver is able to detect a limited number of errors and autocorrect those errors without requiring retransmission.

On the other side, the sender calculates its rate based on the receive report that arrives from the receiver, if this report is not received within two times the maximum interval WebRTC congestion mechanism will consider that all packets during that period have been lost halving the rate in the sender. In order to improve response in lossy environments we could consider calculating the optimal value for this interval considering all the possible situations. Considering the congestion algorithm in

WebRTC [26], the rate should not vary when having between 2-10% of packet losses. Table 3 proves that this mechanism is not working properly as we are noticing reduction of rate with 5% of packet losses, the mechanism should start modifying the rate above 10% of packet lost calculating a new sender available bandwidth (A_s) using Equation 4 being p the packet loss ratio.

$$A_s(i) = A_s(i-1) \times (1 - \frac{p}{2}) \tag{4}$$

If the packet loss is less than 2% the increase of bandwidth will be given by Equation 5.

$$A_s(i) = 1.05 \times (A_s(i-1) + 1000) \tag{5}$$

6.1.4 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.

Delay modeling for real time applications is difficult and can be done using the timestamp of the incoming packets, the incoming frame will be delayed if the arrival time difference is larger than the timestamp difference compared to its predecessor frame.

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.

Table 4 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.

	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 4: Summary of averaged bandwidth with different delay conditions.

We can also observe that every iteration follows a different pattern even having an averaged result, Figure 25 show the test performed at 200ms and the iterations that fail to keep a constant rate making the amount of artifacts in the video affect the quality of the call. We can certainly confirm that the methods that WebRTC should use to control the congestion in the call are not working as they should.

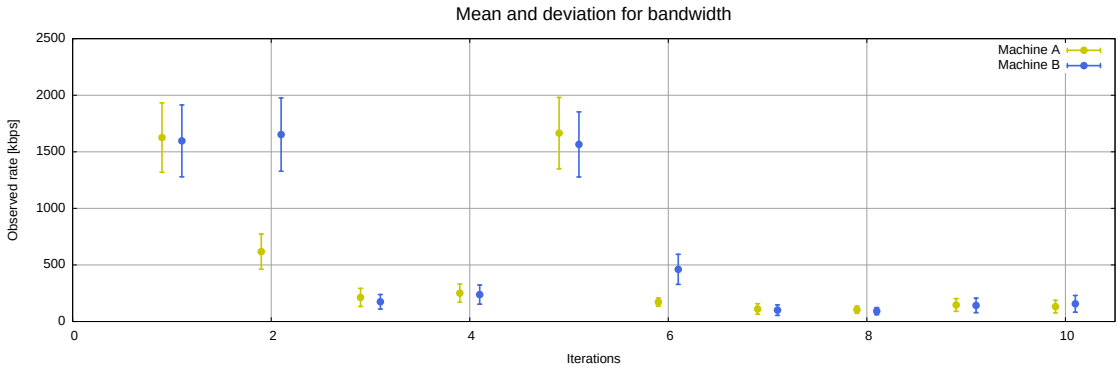


Figure 25: Mean and deviation for P2P 200 ms delay test.

The problem with WebRTC relies in the usage of RTP over UDP for packet transport as UDP does not carry congestion control mechanisms that TCP does, when having real time media adapting the encoding to accommodate the varying bandwidth is difficult and cannot be done rapidly.

Low latency networks will play a big role when WebRTC extends to mobile devices and the ability to react properly to delays and packet losses will be crucial for the success of WebRTC in those environments against its competitors.

6.1.5 Loss and delay

Regarding P2P scenario we also tested the possibility of having a combined lossy network with delay added to it, this kind of environment could be easily found in mobile applications in low coverage areas. We have set 10% packet loss with different delays such as 25ms, 50ms, 100ms and 200ms. In Table 3 we saw an average of over 1 Mbit/s of bandwidth usage in 10% loss environments, the result when adding delay to the constraint is an average of barely 60 Kbit/s. Those results differ due to the difficulty of WebRTC to handle congestion in those environments.

	Machine A	*Machine B*	*Overall*
25ms (Kbit/s)	72.59±18.54	70.69±18.09	71.78±18.32
50ms (Kbit/s)	59.7±16.84	60.36±18	60.03±17.42
100ms (Kbit/s)	63.3±19.29	64.82±20.95	64.06±20.12
200ms (Kbit/s)	66.89±20.12	65.66±19.63	66.27±19.87

Table 5: Averaged bandwidth with different delay conditions with 10% packet loss.

Table 5 describes the averaged bandwidth result with not much difference in each situation.If we study the way WebRTC calculates the rate in difficult situations we can see that the sender will establish its decision on the RTT, packet loss and available bandwidth that is estimated from the receiving side using Equation 4 [26]. Obviously the real output differs form the expected by using the formula, the reason is that even the congestion mechanism on WebRTC calculates the rate using Equation 4, the sender rate is always limited by the TCP Friendly Rate Control (TFRC) formula that is calculated using delay an packet loss ratio together [35].

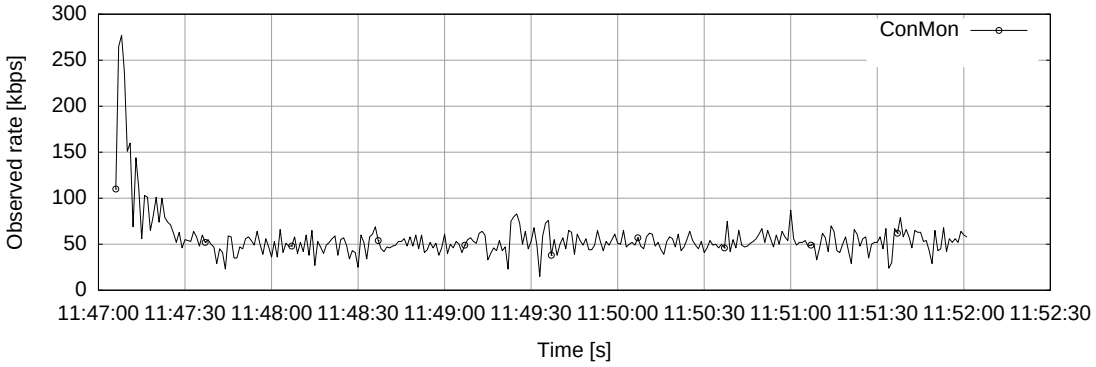


Figure 26: Remote stream bandwidth for 10% packet loss rate and 50ms delay.

Figure 26 is an example that illustrates how the rate is lowered after the beginning of the call even the bandwidth is available. This is due to the formulas and mechanisms previously described.

Carrying delay and losses in the same path will not be handled by the congestion mechanisms in WebRTC giving a low rate output for the stream.

Another interesting factor around this test is the setup time that increases to up 4.6 seconds with 200ms delay and 3 seconds with 50ms, obviously this increase will also affect mobile developers when establishing calls in delayed environments.

6.1.6 Bandwidth and queue variations

We have also performed a different set of tests modifying the bandwidth and queue length. For this part of the test we have chosen to run 500 Kbit/s, 1, 5 and 10 Mbit/s with different queue sizes ranging from 100 ms, 500 ms, 1s and 10 s. In total we have run 12 different tests with ten iterations each.

The queue size is set in slots to *Dummynet* considering each slot as standard ethernet packet of 1500 Bytes, to calculate this number we use Equation 6.

$$\frac{Bandwidth(Bits)}{8 \times 1500} \times Queue(seconds) \tag{6}$$

We have seen a good response when having big queue sizes but larger deviation in bandwidth when reducing this queue size to 100 ms or 500 ms, this produced high delays over 20 ms for every call with different distribution curves. The delay that is given to the duration of the call is not stable and will affect the media flow,

this increasing curve of delay distribution is given by the small queue size which produces bursty packets to arrive to the peer having different delay conditions.

When we tested the 5 Mbit/s case we got a high delay output even the bandwidth response adapted to the constraints, delay deviation is also high and will affect the time the packets arrive with large jittering.

We will study the result of the test performed at 1 Mbit/s limitation as the maximum standard bandwidth for WebRTC is approximately 2 Mbit/s, when having 1 Mbit/s limitation WebRTC will need to adapt the actual encoding rate and bandwidth control to that amount.

Figure 29 represents the bandwidth and mean plotted for all the different tests performed in the 1 Mbit/s case. We can see that the response varies in small amount of bandwidth but with large deviation, when having 500ms and 1s queue size (29c) we have much more deviation in means of packets being buffered in the relay. Otherwise, when the queue size reduces to 100ms (**??**) the deviation gets smaller but delay response is worst.

We can compare Figure 30 delay distribution results for the best case (30a) and worst case (30d). The delay response with large queue is better due to the rapid increase of packets that carry small delay, for the 100ms queue the curve is smoother having packets ranging in all values of delay between 0 and 130ms.

Delay experience with small queue sizes will be worst in the sense of the call flow, we might experience sudden delay situations that WebRTC won't be able to handle, when having larger queue sizes we son't notice the delay variations as much as with the previous example. Having a curvy increase in delay distribution figure will result in sudden delay variations in the call. The conclusion is that WebRTC is able to adapt to low capacity networks using its codec mechanism at the same time as it should improve the congestion control systems to adapt to different buffer sizes and queuing conditions.

The congestion mechanisms in WebRTC will stabilize the rate until the amount of delay triggers the rate change to fit the new queue state requirements. Figure 27 and 28 show the bandwidth and delay for the same stream and how the rate adapts once the queues are full increasing the delay on the packets, rate is lowered and queues get empty giving producing low delay.

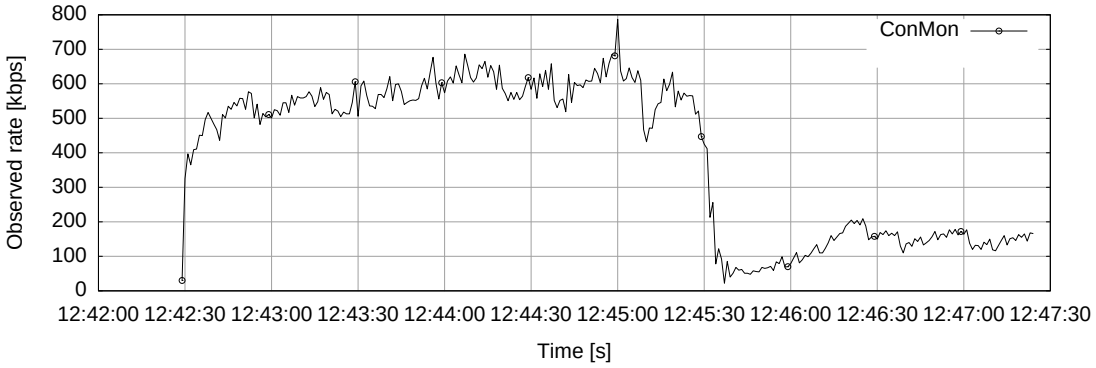


Figure 27: Remote stream bandwidth for 1 Mbit/s and 500ms queue size.

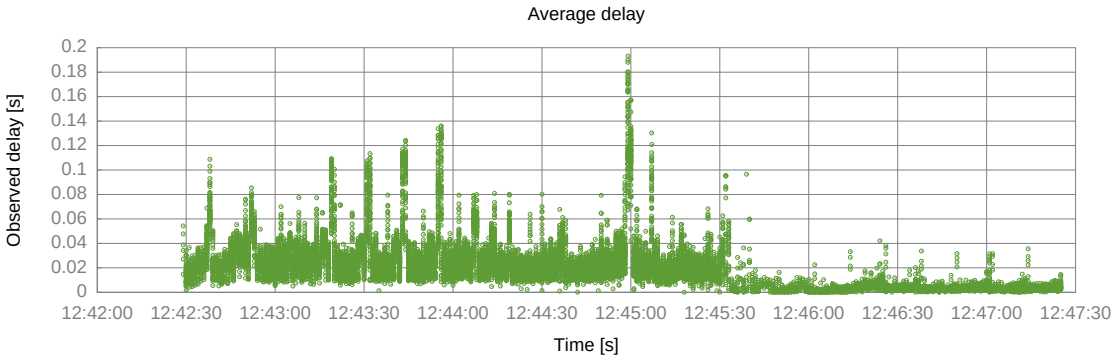
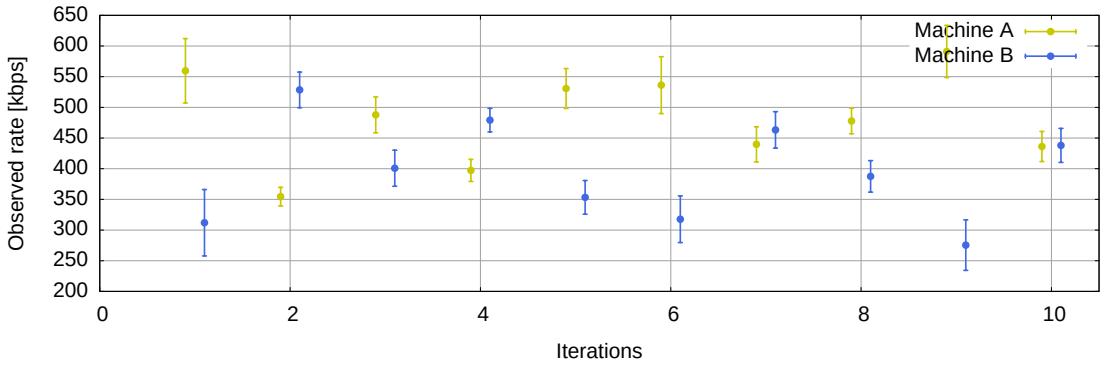
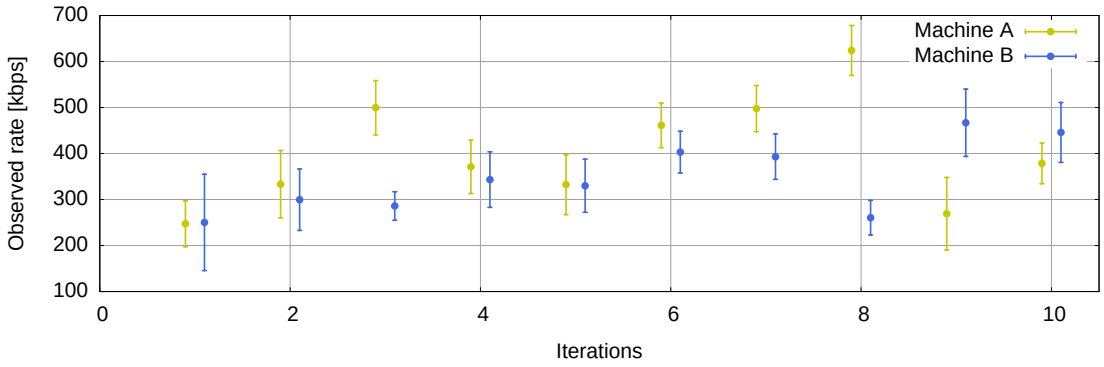


Figure 28: Stream delay for 1 Mbit/s and 500ms queue size.

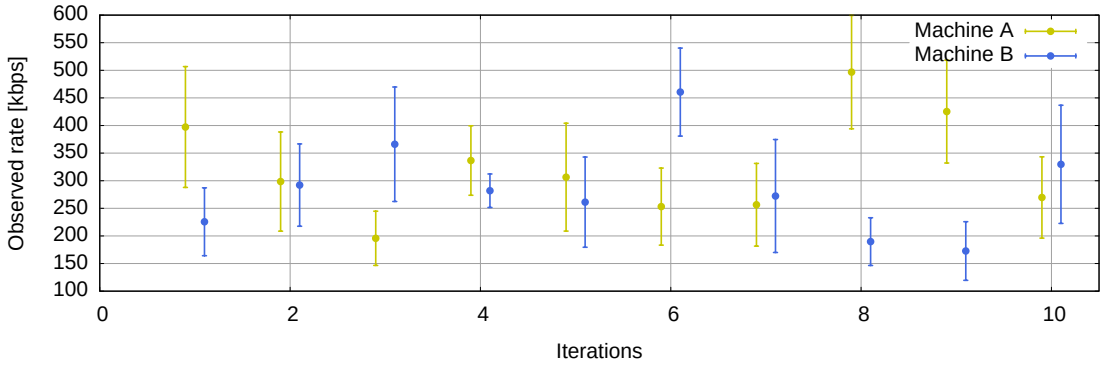
Studying the way the sender takes decisions about rate constraints we can observe that the available bandwidth estimates calculated by the receiving side are only reliable when the size of the queues along the channel are large enough [26] . When having short queues along the path the maximum usage of the bandwidth cannot be estimated if there is no packet loss in the link, as in this case the packet loss is negligible, the connection is not able to use the maximum amount of bandwidth available in the link.



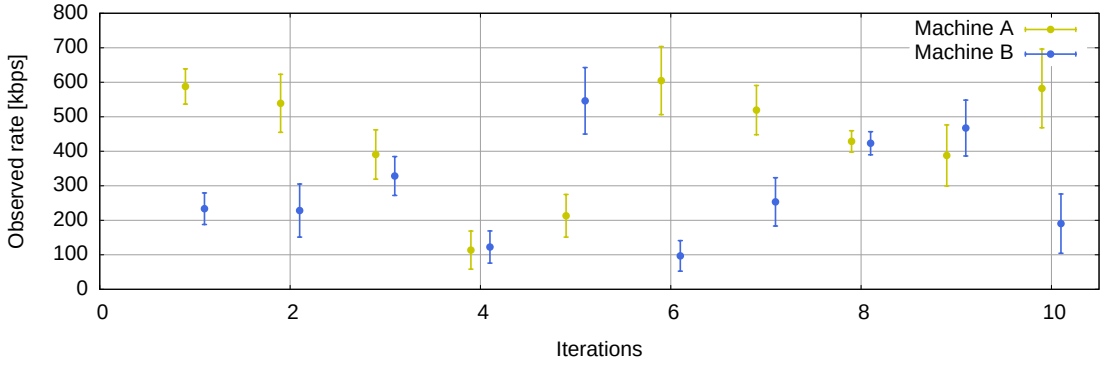
(a) 1 Mbit/s and 10s queue size.



(b) 1 Mbit/s and 1s queue size.

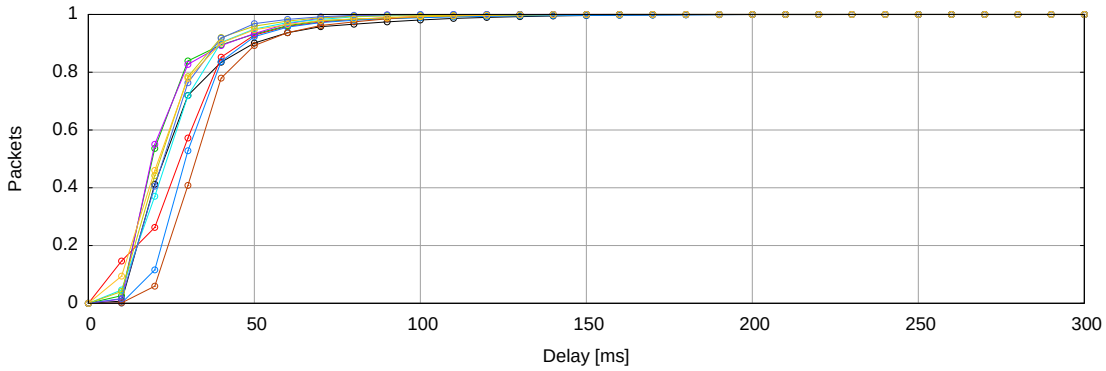


(c) 1 Mbit/s and 500ms queue size.

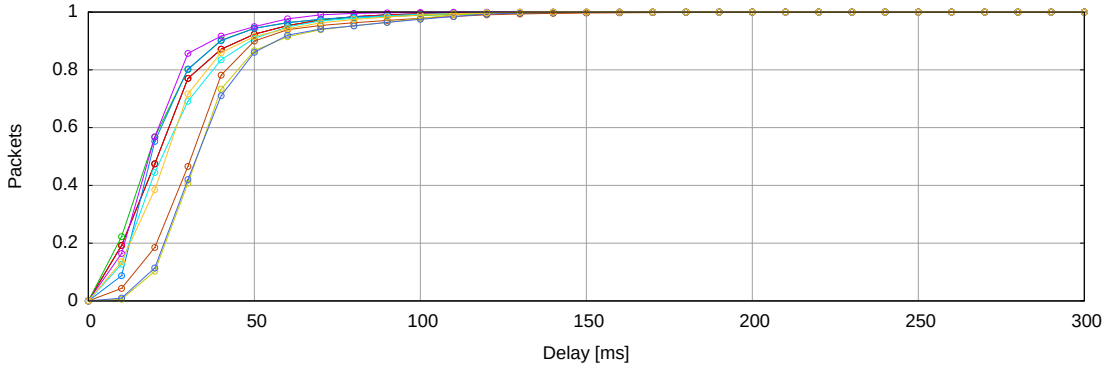


(d) 1 Mbit/s and 100ms queue size.

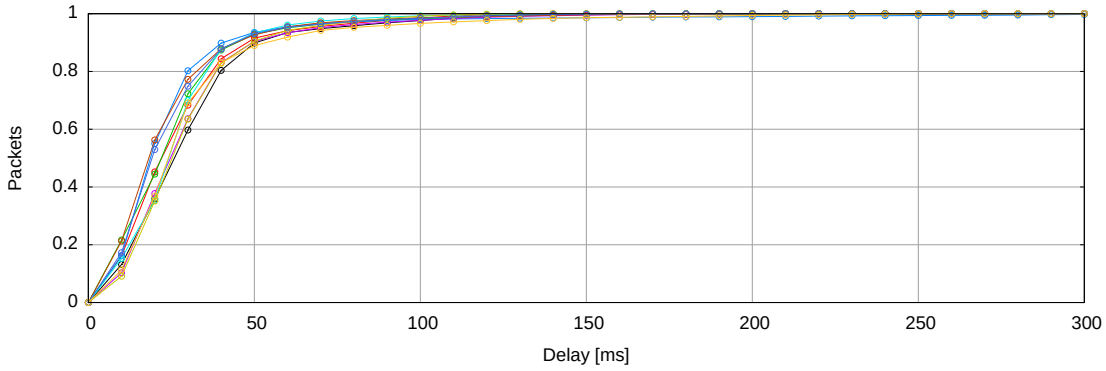
Figure 29: Bandwidth and mean for 1 Mbit/s with multiple queue sizes



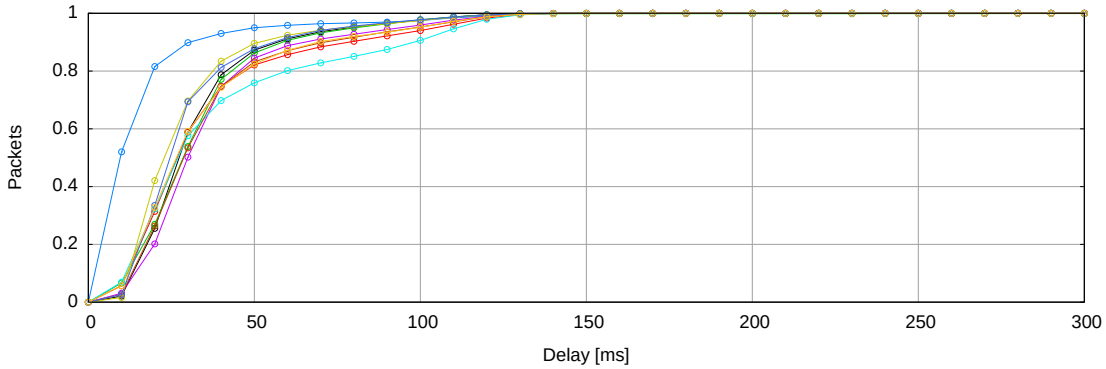
(a) 1 Mbit/s and 10s queue size.



(b) 1 Mbit/s and 1s queue size.



(c) 1 Mbit/s and 500ms queue size.



(d) 1 Mbit/s and 100ms queue size.

Figure 30: Delay distribution for 1 Mbit/s with multiple queue sizes

6.2 Loaded network

Similar to the previous test, in this case we will be measuring the performance of WebRTC in a loaded network using a tool named *Iperf*. This tool will allow us to emulate traffic between our two peers loading the network according to our needs with UDP or TCP packets. The configuration we will use is the one shown in Figure 31 with the clients running *Dummynet* instead of the relay. This scenario is chosen due its widely usage in real devices, having video calls meanwhile manipulating large amounts of online data is something that might happen when using WebRTC.

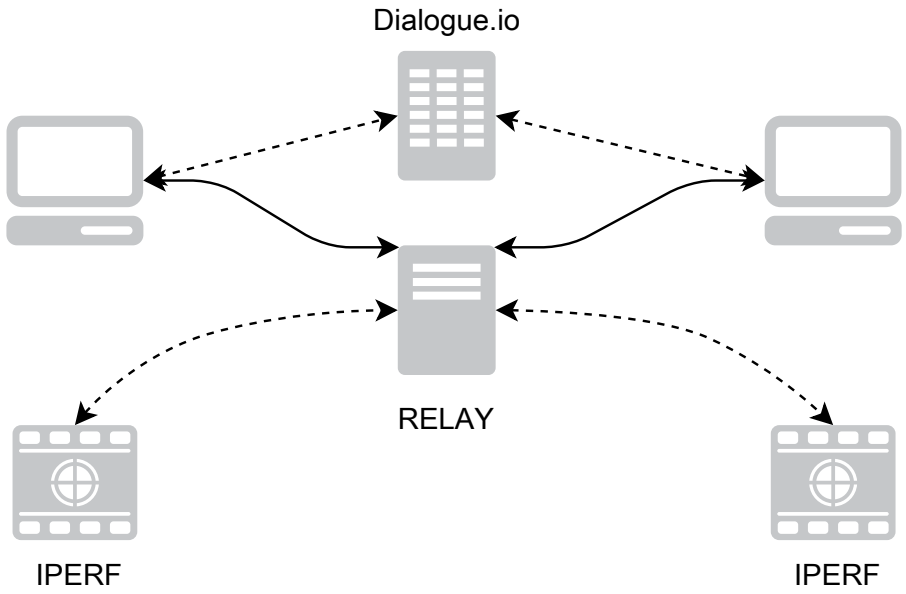


Figure 31: Topology for traffic flooded path using *Iperf*.

In this scenario we are interested in measuring also the behavior of real bandwidth setups for different environments, we will be testing the link with 100/10 Mbit/s and 20/4 Mbit/s limitations, the second one could be defined as the standard for HSPA networks. The data that will be sent to the other peer will be either 10 Mbit/s of TCP and UDP traffic or 2 Mbit/s.

First we will run the server as daemon on the recipient of the packets by executing:

```
# iperf -s -D
```

The next step will rely on the usage of UDP or TCP, *Iperf* sends TCP packets by default, to do so we will run:

```
# iperf -c XXXX -t 300 {-u} -b 10m/2m
```

In the previous command, *-t* is the amount of time the test length, *-c* is the feature that configures the remote server to send the packets to, *-u* is going to be

used to sent UDP datagrams instead of TCP and *-b* will define the amount of Mbit/s to be sent to the remote server. In this case every test is run three times.

Table 6 summarizes the results of the 10 Mbit/s TCP packet test without *Dummynet* constraints in the link.

	Machine A	*Machine B*	*Overall*
CPU (%)	81.06±5.2	82.15±5.23	81.16±5.22
Memory (%)	35.65±0.43	34.27±0.39	34.96±0.41
Bandwidth (Kbit/s)	990.11±202.62	1250.13±264.38	1120.12±233.506
Setup time (ms)	1533.66±11.3	1577.66±41.89	1555±32.59
RTT (ms)	25.61±16.02	24.76±14.11	25.19±15.07
Delay (ms)	81.61±11.42	83.99±11.42	81.61±11.42

Table 6: IPERF 10 Mbit/s TCP test without link constraints.

The bandwidth rate in the call is affected by the traffic of TCP packets along the path, at the same time we are getting higher delays. Call behavior in this environment changes in every iteration being unpredictable, Figure 32 represents the bandwidth mean and deviation of every iteration, we can easily observe that in the worst case we are getting three times less rate than the optimum case, ranging from 1.5 Mbit/s to under 400 Kbit/s in the worst iteration.

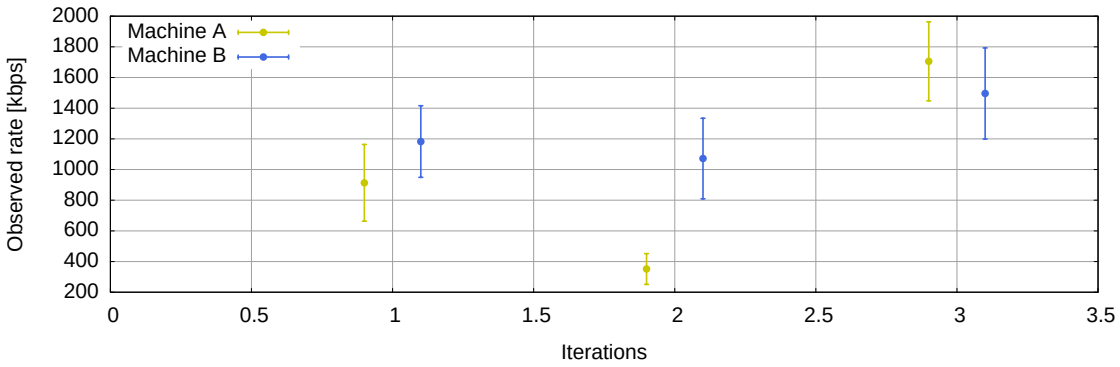


Figure 32: Bandwidth mean and deviation for 10 Mbit/s TCP *Iperf* test without link constraints.

We can observe some interesting behavior in all three iterations when looking at Figure 33 total delay distribution, the response varies from all three tests being all of them bad, a lot of sudden delay changes will appear during the call making real time communication difficult. The delay deviation is small but the tolerance for TCP flooded networks is low int WebRTC.

Now we will test the behavior when sending those 10 Mbit/s with UDP and TCP in a constrained link of 100/10 (downlink/uplink), in this test *Dummynet* scripts have been executed on the client side instead of in the Relay. Table 7 shows

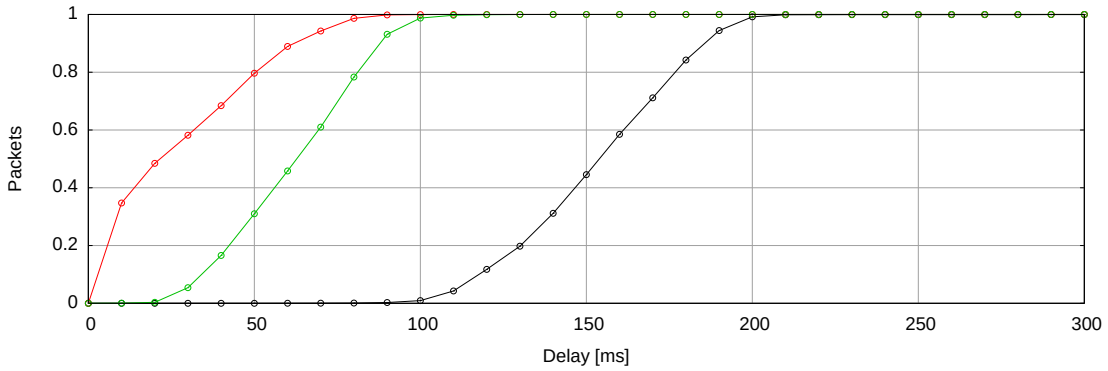
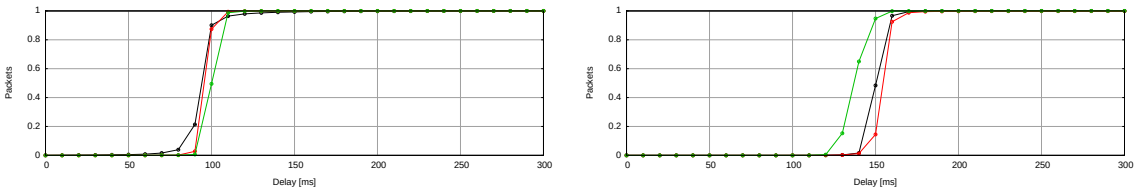


Figure 33: Total delay distribution for 10 Mbit/s TCP *Iperf* test without link constraints.



(a) Delay distribution response for UDP test

(b) Delay distribution response for TCP test

Figure 34: 10 Mbit/s UDP/TCP *Iperf* test with 100/10 link condition.

the different bandwidth responses between TCP and UDP traffic, in both cases the link constraint have been the same but the result varies. We will se an increase of rate with TCP flooded packets but also an increase of delay, this delay might be produced due the need of processing more packets with TCP than the simple mechanism of UDP.

	Machine A	*Machine B*	*Overall*
Bandwidth UDP (Kbit/s)	159.41±28.69	149.04±25.76	159.23±27.23
Delay UDP (ms)	98.07±3.14	98.85±2.75	96.85±2.94
Bandwidth TCP (Kbit/s)	208.97±20.64	194.41±18.9	201.69±19.77
Delay TCP (ms)	146.9±4.38	147.92±4	147.41±4.23

Table 7: IPERF 10 Mbit/s TCP and UDP test with constrained 100/10 Mbit/s link.

Delay distribution response in a constrained environment (Figure 34a and 34b) is smoother compared to Figure 33, the absolute amount of delay is larger but the distribution curve is better for WebRTC needs as it does not have any sudden increase of delay. Delay response when having constraints will output a better delay distribution but with higher RTT in the link.

When testing the 2 Mbit/s TCP and UDP flows with 20/4 Mbit/s constraints

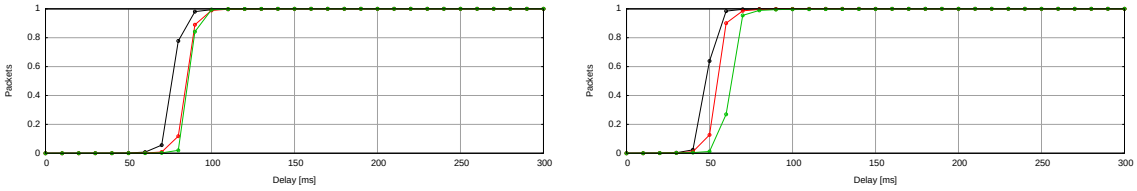
results are surprisingly close to the version without constraints, we are testing this configuration due to its similitudes to HSDPA networks that carry a similar averaged bandwidth. Unstable bandwidth is also noticed in this test but values for the rate are much higher and delay distribution graphs are similar to Figure 34. We are using 2 Mbit/s flows to imitate the encoding rate for an online streaming 1280x720 HD video.[1]

Table 8 describes the output we had in terms of rate and delay for the 2 Mbit/s test in a HSDPA type network. Rate adaptation is good even having an small uplink capacity of 4 Mbit/s, the way the rate is adapted to this link confirms that in this kind of not delayed or lossy low latency networks WebRTC could perform properly with simultaneous ongoing traffic.

	Machine A	*Machine B*	*Overall*
Bandwidth UDP (Kbit/s)	683.81±259.38	749.66±249.69	716.74±254.53
Delay UDP (ms)	56.34±2.83	54.31±2.64	55.32±2.74
Bandwidth TCP (Kbit/s)	760.94±238.44	1174.95±235.12	967.94±236.78
Delay TCP (ms)	85.18±2.3	80.04±2.26	82.61±2.28

Table 8: IPERF 2 Mbit/s TCP and UDP test with constrained 20/4 Mbit/s link.

From the delay distribution point of view (Figure 35), the output is similar in both tests being TCP slightly better (35b) with less absolute delay and with an acceptable variation.



(a) Delay distribution response for UDP test (b) Delay distribution response for TCP test

Figure 35: 2 Mbit/s UDP and TCP *Iperf* test with 20/4 link condition.

In general, the response of WebRTC congestion mechanisms with ongoing link traffic should be better as this environment will be common for all users. The bandwidth mechanism produces an acceptable call rate but should produce delays smaller than one second which are acceptable from the usability perspective, the delay distribution for the standard case with an ongoing traffic of 10 Mbit/s is not as good as expected but it might be due to the high capacity on the path and the way *Iperf* simulates the traffic.

[1] http://www.adobe.com/devnet/adobe-media-server/articles/dynstream*l*ive/*popup.html*

6.3 Parallel calls

In this part of the test we will be checking how WebRTC handles multiple parallel calls with different peers, this is not to me mixed with mesh style of topology as it will be running using different tabs or processes through the same TURN path. Figure 36 represents the topology used for the test.

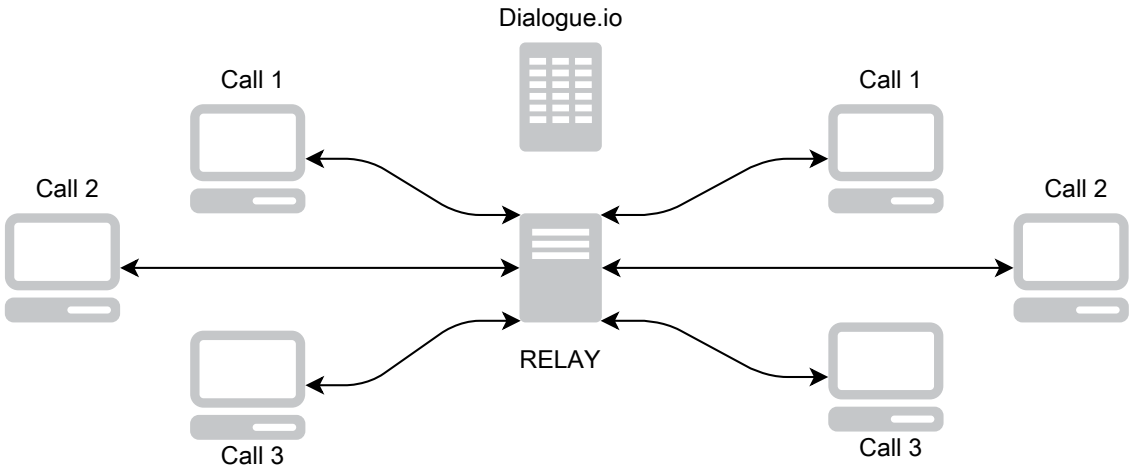


Figure 36: Topology for three different parallel calls using the same link.

We will run a combined batch of tests using 2 and 3 simultaneous calls without *Dummynet* or with 20 Mbit/s and 10 Mbit/s bandwidth limitation for the link. The case without any constraint will run with the standard 100 Mbit/s of the ethernet link capacity. For the test we have focused in running the calls in the same machine but in different processes.

This kind of environment will be given in local networks or it could be compared with mesh topologies handling multiple peer connections, from the resources perspective it will be interesting to observe the CPU and memory consumption as every PeerConnection will be working in a different process, the machine used carries 1 CPU and 2 Gb of RAM.

	CPU (%)	*Memory (%)*
Three calls	99.25±2.41	44.99±0.5
Two calls 20 Mbit/s	95.67±3.51	46.16±0.37
Two calls 10 Mbit/s	86.83±5.03	44.91±0.32
Two calls	81.6±6.48	42.61±0.35

Table 9: Memory and CPU consumption rates for parallel calls in different link conditions.

Table 9 describes the resource comparison between two and three simultaneous calls. CPU usage is critical when handling three peer connections or when

the network condition forces the congestion mechanism to continuously adapt the bandwidth and encoding. In this test, each call is placed in a different process which should improve the results as the OS will handle them better than in a single thread. When the CPU load gets to its maximum the performance of WebRTC for encoding/decoding and transmission is deprecated, in this kind of topologies having high CPU performance increases the call quality.

	Machine A	*Machine B*	*Overall*
Three calls	768.04±180.93	850.1±223.84	809.07±202.38
Two calls 20 Mbit/s	432.56±141.32	531.13±169.82	481.85±155.56
Two calls 10 Mbit/s	178.83±60.05	141.83±42.02	160.24±51.04
Two calls	392.08±181.9	545.94±259.27	469.01±221.09

Table 10: Bandwidth rates for parallel calls in different link conditions.

The bandwidth represented in Table 10 is the one used per each machine with the calls together, we will study the worst case. The bandwidth for three calls is to use an average of 800 Kbit/s with different responses in every case, the deviation is approximately ±202 Kbit/s.

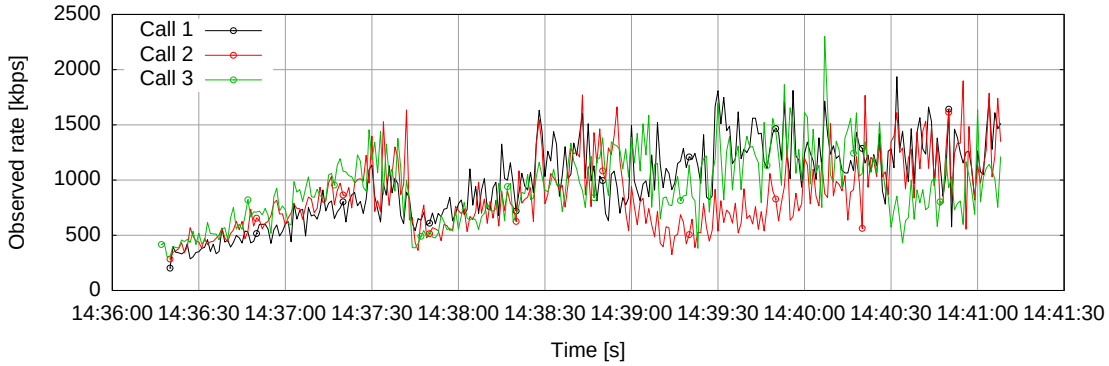
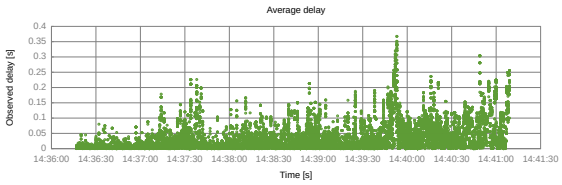
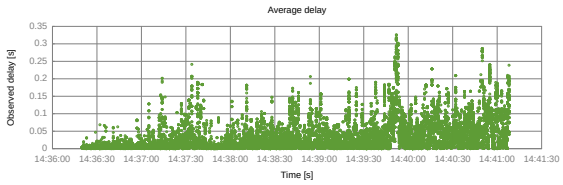


Figure 37: Bandwidth representation for all remote streams in a synchronous three peer parallel call for first iteration.

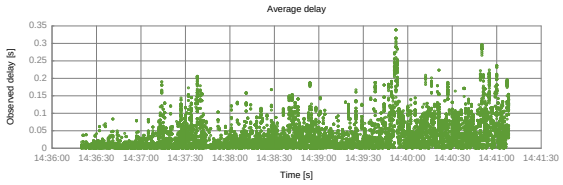
Furthermore, it is interesting to see the global rate on the call as the bandwidth averaged is not following a stable value, Figure 40 represents the bandwidth during all call for the remote video stream of the three peers. We can see how the rate mechanism tries to use the maximum available rate for the actual video encoding but fails to reach the 2 Mbit/s as multiple calls are running and behaving in the same way, this decision is taken also considering the delay that limits the maximum available rate for the call. Figure represents the delay on the same streams during the call, we can observe those peaks of delay during the same period as the bandwidth rise, the result of this is the sudden drop of bandwidth, this mechanism is triggered multiple times.



(a) Remote stream for call 1



(b) Remote stream for call 2



(c) Remote stream for call 3

Figure 38: Delay representation for all remote streams in a three peer parallel call.

We can compare this scenario with the one in Figure 27, the difference relays in the channel condition, in the example of Figure 27 the channel condition set high restrictions on the path making the rate drop and keep stable as the condition didn't change after that moment. In Figure 38, path condition changes after the drop as it becomes available again as all the three peer calls behaved the same way.

In general the delay response in all the calls is bad, Figure 39 plots the delay distribution of the three simultaneous calls, the slow increase of delay makes the call lag large and variable, probably the user experience for all the three calls will be bad.

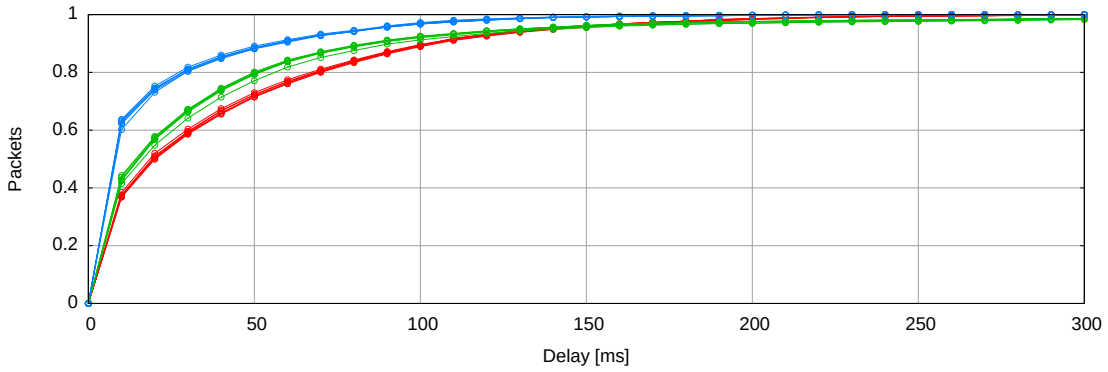


Figure 39: Total delay distribution for three parallel calls.

The problem relies on the separate treatment of every peer connection, they have no acknowledge of having different similar processes going so the rate change cannot be constrained by the other ongoing calls. Notice that for this test all calls started at the same time, for this purpose we ran a second set of tests starting every call delayed by 15 seconds to check the behavior of the system.

After analyzing the captures of the new asynchronous call we can plot the averaged bandwidth to be approximately 1154 Kbit/s with ±250 Kbit/s of deviation.

Overall averaged results are significantly higher than in the previous case but we should have a close look at Figure that represents the bandwidth behavior of the three calls during all the duration for the first iteration we can observe that the average is high but one of the calls rate is very low.

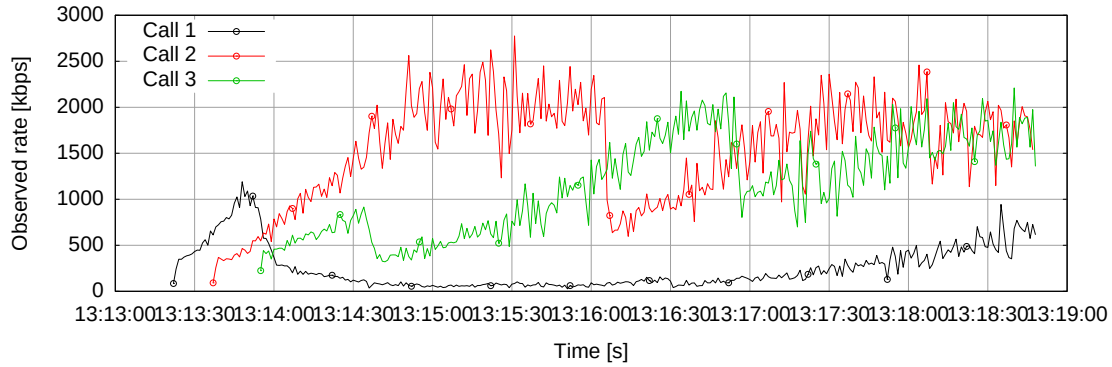


Figure 40: Bandwidth representation for all remote streams in an asynchronous three peer parallel call for iteration one.

In this case, the first call that started to increase its rate suddenly drops it to approximately 100 Kbit/s and stays there along the call meanwhile the other two parallel calls try to obtain the maximum available bandwidth of the path. This environment is more approximated to the a real scenario as users won't start calls exactly at the same time but they will probably do that randomly, some calls quality will be degraded with barely no quality and others will have sudden drop of bandwidth affecting the interaction with the user.

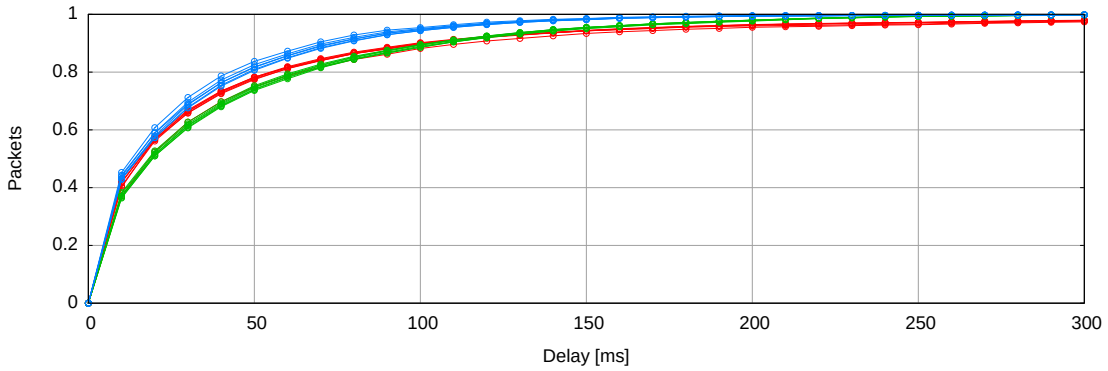


Figure 41: Total delay distribution for three asynchronous parallel calls.

Figure 41 represents the delay distribution for the asynchronous test, similar to the previous example (Figure 39) but with a worst delay response. Delay will affect to the user experience having sudden cuts of communications of milliseconds that will occur randomly, they are not large but they are random and unexpected.

WebRTC should be able to identify parallel connections of the same type and balance the bandwidth usage, this is difficult as the transport level uses already

existing RTP technology over UDP, the conclusion is that WebRTC will not be reliable for multiple parallel calls in an average computer.

6.4 Mesh topology

A common setup in real-time communications is video conferencing, this way of calling people is widely used in virtual meetings. Until this moment there were multiple available options in the market, with the arrival of WebRTC this feature extends to the web application world, with multiple options and features to be enabled with it. We will try to determine if WebRTC is mature enough to handle multiple peers at the same time and session, the way this is done varies depending on the technology, we will study pure peer-to-peer mesh networks.

The most common option for this kind of environments is to use an MCU to perform the relying of the media through a unique connection by multiplexing the streams in a single one. There are some MCU available in the market for WebRTC but the API is still not evolved enough to allow multiplexing of streams over the same Peer Connection, in the following updates of the API this should be enabled allowing developers to perform real media multiplexing. Some vendors offer MCUs that require extra plugins to be installed, this is due to the impossibility to multiplex multiple media streams over the same Peer Connection, this is required when using Google Hangouts product.

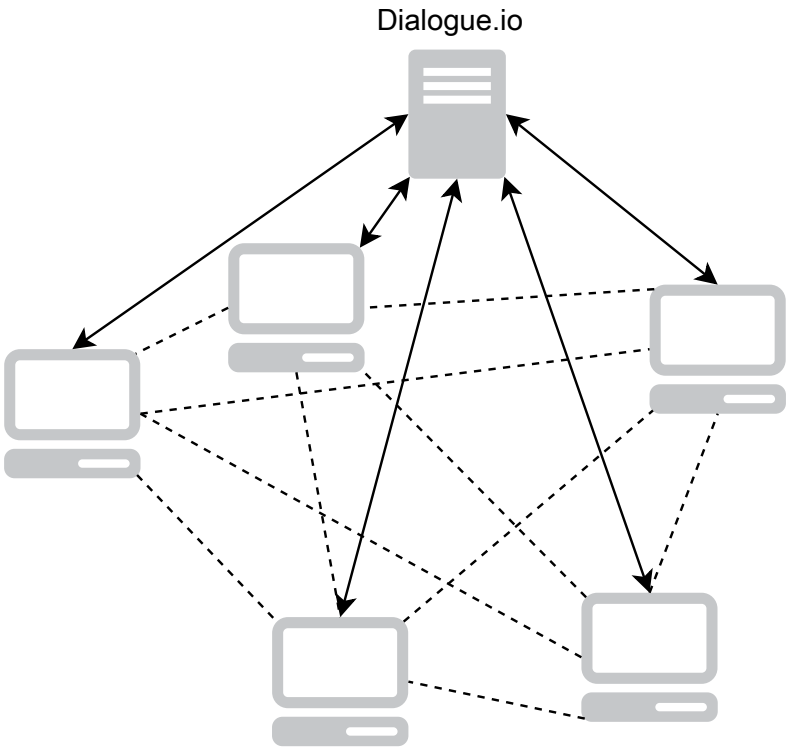


Figure 42: Mesh topology for WebRTC.

Our topology, shown in Figure 42, consist in a mesh network of different virtual

machines that connect by using our centralized *dialogue.io* application for signaling, media is sent over the peers directly, this will produce a big load in the performance for those clients as the amount of peer connections will be large, each of them obliged to encode and decode media, different from the previous example of parallel calls in this case all the peer connections will be running in the same process making the resource management a key point for the performance of WebRTC. We have increased the amount of CPUs to two in order to get proper results.

Three peers are used for the first test and it will increased by one peer every test, the output for the first test is shown in Table 11. It is important to consider the amount of time required to set up the call, the worst result for the setup time in this scenario has been of 2206 ms to set up the three whole mesh.

	Machine A	*Machine B*	*Machine C*
CPU (%)	88.5±4.77	89.49±4.46	91.65±4.23
Memory (%)	49.25±0.33	50.52±0.28	55.52±0.29
Bandwidth (Kbit/s)	333.38±115.13	344.48±95.43	410.77±115.97

Table 11: CPU, memory and bandwidth results for three peer mesh scenario without relay.

CPU and memory usage is nearly double than in the previous scenario, considering we have doubled the CPU capacity, being this very consuming taking into consideration that is the only application running on the test machine.

Bandwidth will be an important constraint when having multiple peer connections, in a single P2P call we have two streams that are being sent/received, in mesh this amount is greater. Figure 43 represents the bandwidth mean and deviation for the three peer mesh call without relay, for every machine we will have two remote streams plotted with different bandwidth rates for them, is interesting to see how they are related in being one always greater than the other, this means that one stream will handle better quality than its other peer.

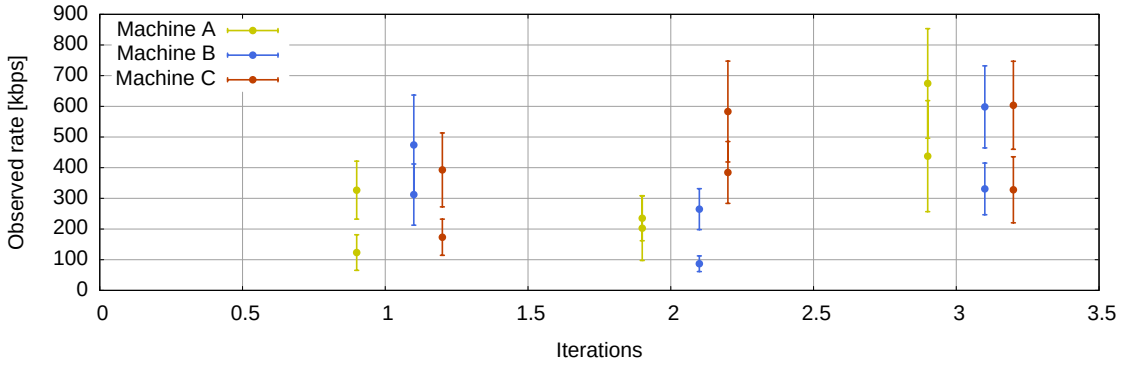
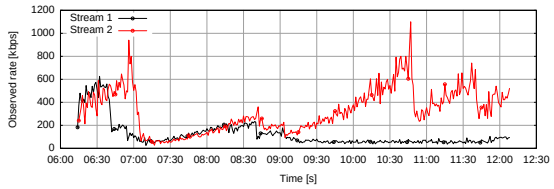


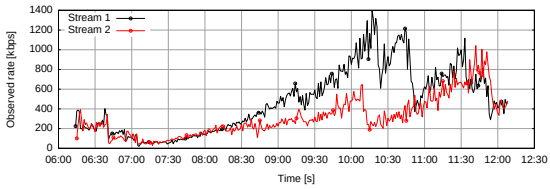
Figure 43: Bandwidth average and deviation for three peers mesh call.

The total averaged bandwidth is approximately 362 Kbit/s, as it can be observed

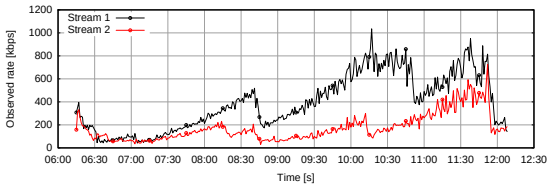
this value does not agree with the real plot for every peer (43), this is due to the disturbances of the call and the continuous rate adaptation of the congestion mechanism. For better accuracy and understanding of what happened during the call we should look directly to the continuous rate of the incoming streams in Figure 44.



(a) Remote streams for machine A



(b) Remote streams for machine B

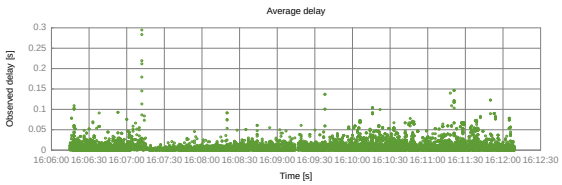


(c) Remote streams for machine C

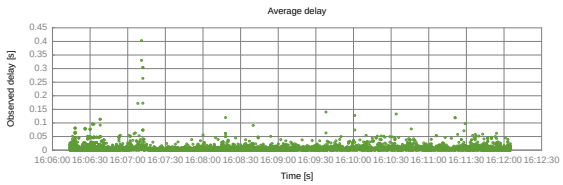
Figure 44: Bandwidth plot during all the call for the incoming streams of each peer for the first iteration.

Each machine will have a pair of incoming streams, the rate calculated for the streams is based on the same mechanisms as the previous tests, this figure should look similar to Figure 40 as they both are running multiple peer connections in the same test, we can see how the rate is being recalculated and how one of the streams is always being deprecated to a very low rate compared to the rest at a certain point when the other increase their throughput. This response of the rate adaptation will be negative to the call.

Furthermore, a different behavior is observed in the delay of the call (Figure 45), compared to the parallel calls (40) which had a very bad response to OWD in this specific case the output for the same four streams in delay is much better than the prior test.



(a) Delay for Figure 44a stream 1



(b) Delay for Figure 44a stream 2

Figure 45: Delay output for Figure 44a incoming streams.

The global delay distribution response is also good compared to previous tests, we can compare the obtained one in Figure 46 with the three parallel calls in Figure 39 and 41. Considering the curve of the delay for the mesh networks we can say that the delay won't be affecting significantly the user experience during the call duration, we

won't have good quality regarding the video and rate but there won't be any sudden delays expected in the communication. This means that from the perspective of a non relayed mesh call we can have three peers with relative acceptable bandwidth in most of the streams with an acceptable delay, the only drawback observed is the amount of used resources by the process, considering that the browser was the only application running on the test machine increasing the amount of processes will probably affect the behavior of the call.

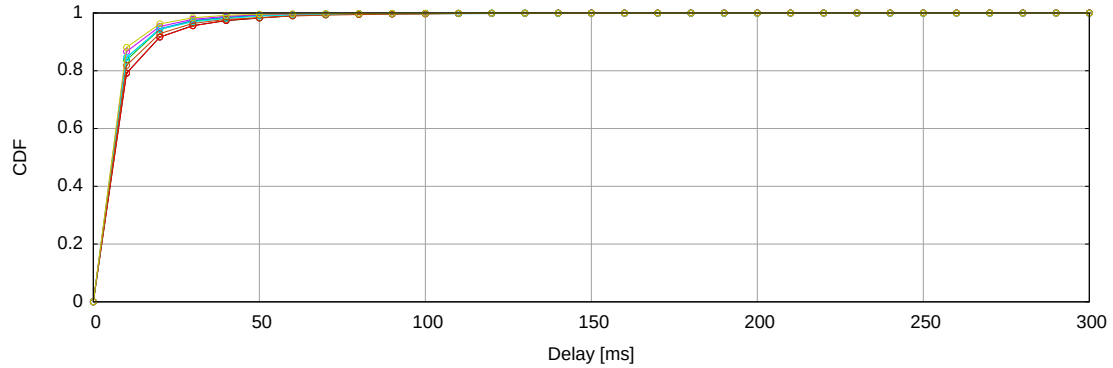


Figure 46: Total delay distribution for three peer mesh call without relay.

Similar to the previous scenario we have run a test using the TURN as relay for the media to check the results, the bandwidth and CPU results are approximately the same, Figure 47 represents the averaged delay result for the three iterations in both tests. Results are slightly better with the TURN, this might be due to the fixed path for routing the packets that produce smaller deviation, the averaged delay is similar in both cases.

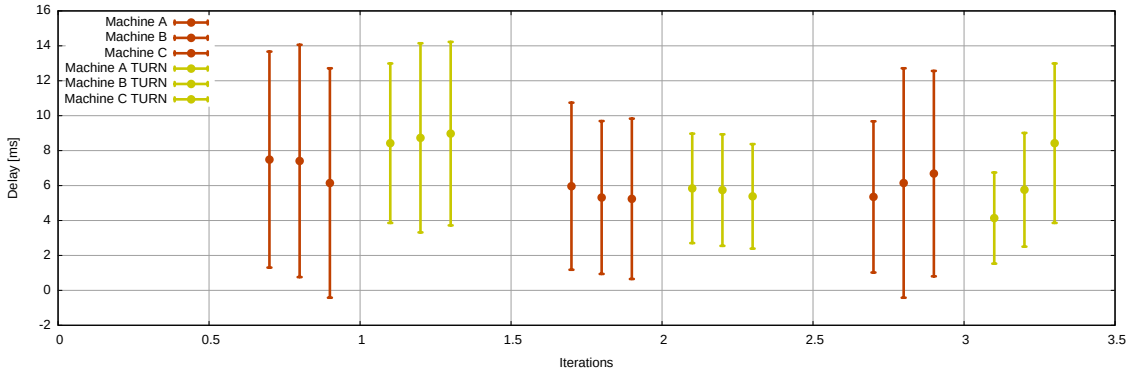


Figure 47: Averaged delay and deviation for TURN and non relayed mesh call for all iterations.

For environments such as mesh, the usage of MCU should be considered as a good alternative to the pure P2P option, this could drastically reduce the resource consumption and bandwidth distribution on the different peer connections when using multiple peer topologies.

6.5 CPU performance

After running all the tests we can see that WebRTC is facing a big problem when it comes to CPU and memory consumption. Averaged computers could have problems when handling multiple peer connections in WebRTC, we have also tested the CPU usage in all those different scenarios.

6.6 Summary of results

After all the performed tests we can conclude that WebRTC is a young protocol for real time communication that still has long way to go in terms of congestion control to adapt to all topologies, still is a reliable and interesting method that uses all existing technologies previously developed for other projects there is a lot of work to do in terms of scalability and environment adaption.

Considering the existing WebRTC congestion mechanisms we can conclude that is still not ready for low tolerance networks or to be used simultaneously in different calls, the resource consumption of the existing WebRTC API is still something to care if we want to integrate WebRTC in mobile devices.

In multiple peer connections environments the usage of an MCU is something that should be considered in order to reduce load and increase performance, the actual WebRTC API does not allow to natively use multiplexing of streams over the same peer connection but the roadmap of the standardization protocol includes this feature for future versions of WebRTC.

Delay response for some low latency environments is not suitable for production applications and scenarios should be considered by developers prior deciding to use WebRTC instead of other competitors.

7 Conclusion

The end.

References

[1] NetMarketShare. Market Share Statistics for Internet Technologies. http://www.netmarketshare.com/.

[2] 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.

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

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

[5] 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.

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

[7] 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.

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

[9] 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.

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

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

[12] 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.

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

[14] 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.

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

[16] J. Uberti and C. Jennings. Javascript Session Establishment Protocol. http://tools.ietf.org/html/draft-ietf-rtcweb-jsep, October 2012.

[17] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). http://tools.ietf.org/html/rfc5389, 2008.

[18] J. Rosenberg, R. Mahy, and P. Matthews. Traversal Using Relays around NAT (TURN). http://tools.ietf.org/html/rfc5766, 2010.

[19] J. Rosenberg. Interactive Connectivity Establishment (ICE). http://tools.ietf.org/html/rfc5245, 2010.

[20] JM. Valin, K. Vos, and T. Terriberry. Definition of the Opus Audio Codec. http://tools.ietf.org/html/rfc6716, 2012.

[21] VP8 Patent Cross-license Agreement.

[22] R. Jesup, S. Loreto, and M. Tuexen. RTCWeb Datagram Connection. http://tools.ietf.org/html/draft-ietf-rtcweb-data-channel, 2012.

[23] R. Stewart. Stream Control Transmission Protocol. http://tools.ietf.org/html/rfc4960, 2007.

[24] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. http://tools.ietf.org/html/rfc6347, 2012.

[25] 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.

[26] H. Alvestrand, H. Lundin, and S. Holmer. A Google Congestion Control Algorithm for Real-Time Communication. http://tools.ietf.org/html/draft-alvestrand-rmcat-congestion, 2012.

[27] 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.

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

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

[30] J. Ott, S. Wenger, N. Sato, C. Burmeister, and J. Rey. Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF). http://www.ietf.org/rfc/rfc4585.txt, 2003.

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

[32] 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.

[33] Creytiv. Restund. http://www.creytiv.com/restund.html.

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

[35] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. http://tools.ietf.org/html/rfc5348, 2008.

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

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

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

[39] 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 [36].

Differences between standard driver and modified 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 [36] 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* [37], 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.[2]

```
# 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.

[2] 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 [38]. 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 [39], 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 [34].

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* and *./ipfw/glue.c*.

```
Index: ipfw/dummynet.c
===================================================================

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);
}
```

The problem arises from a the misassumption of *sizeof(long) == 4* in 64-bit architectures which is false. By changing those two files we are modifying the system in order to accept higher values than 100 for the queue length.

```
Index: ipfw/glue.c
===================================================================

char filename[256];   /* full filename */
char *varp;
int ret = 0;          /* return value */
long d;

if (name == NULL) /* XXX set errno */
        return -1;


        fprintf(stderr, "\%s fopen error reading filename \%s\n",
            __FUNCTION__, filename);
        return -1;
}
if (fscanf(fp, "\%ld", &d) != 1) {
        ret = -1;
} else if (*oldlenp == sizeof(int)) {
        int dst = d;
        memcpy(oldp, &dst, *oldlenp);
} else if (*oldlenp == sizeof(long)) {
        memcpy(oldp, &d, *oldlenp);
} else {
        fprintf(stderr, "unknown paramerer len \%d\n",
        (int)*oldlenp);
}
fclose(fp);


        fprintf(stderr, "\%s fopen error writing filename \%s\n",
            __FUNCTION__, filename);
        return -1;
 }
if (newlen == sizeof(int)) {
        if (fprintf(fp, "\%d", *(int *)newp) < 1)
                ret = -1;
```

```
} else if (newlen == sizeof(long)) {
        if (fprintf(fp, "\%ld", *(long *)newp) < 1)
                ret = -1;
} else {
        fprintf(stderr, "unknown paramerer len \%d\n",
                (int)newlen);
}
fclose(fp);
```

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.

C Scripts for testing WebRTC

Listing 7: Script for testing WebRTC with 15 iterations

```
#!/bin/bash
#

#First argument will define the name of the test, second the video to use
#and third may define the IPERF configuration if used

#We also will use this as test example modifying some parameters for the
#other examples such as parallel and mesh


echo "" > 1to1.log
#Exporting variables required for the test
echo "Exporting variables"
PATH="$PATH:/home/lubuntu/MThesis/v4l2_file_player/"
PASSWORD=lubuntu

#Timers for the call duration and break time after the call
REST_TIMEOUT=30
TIMEOUT=300

INIT_TIME=$(date +"%m-%d-%Y_%T")

#Define folders to sabe files
backup_files="/home/lubuntu/MThesis/ConMon/rtp/rtp_*"
mkdir results/$INIT_TIME"_"$1
dest_folder="/home/lubuntu/results/"$INIT_TIME"_"$1
echo "Starting $INIT_TIME"
counter=0

#Loop the test 15 times to avoid call failures
while [ $counter -le 14 ]
do
        actual_time=$(date +"%m-%d-%Y_%T")
        echo "Iteration - $counter"
        #Clean all ongoing processes from previous iterations
        echo "Cleaning processes"
        echo $PASSWORD | sudo -S killall conmon >> 1to1.log 2>&1
        killall v4l2_file_player >> 1to1.log 2>&1
        killall chrome >> 1to1.log 2>&1
        sleep $REST_TIMEOUT

        #Set virtual device for Webcam
        echo "Setting dummy devices"
```

```
        echo $PASSWORD | sudo -S modprobe v4l2loopback devices=2 >>
            1to1.log 2>&1

        cd MThesis/ConMon
        #Start ConMon and configure 192.168.1.106 which is the turn relay
            for the media
        echo $PASSWORD | sudo -S ./conmon eth3 "udp and host 192.168.1.106"
            --turn >> 1to1.log 2>&1 &
        cd ../..

        #Load fake video into virtual device
        echo "Loading video"
        v4l2_file_player /home/lubuntu/MThesis/v4l2_file_player/$2 352 288
            /dev/video1 >> 1to1.log 2>&1 &
        #If third argument available then we run the IPERF
        if [ $# -eq 3 ]
        then
                        iperf -c 192.168.1.106 -t 300 -i 5 -b $3 >> 1to1.log
                            2>&1 &
        fi

        #Load browser pointing the test site with the n= parameter that
            will define the StatsAPI filename
        #We need to ignore the certificate errors to load the page with an
            untrusted certificate
        DISPLAY=:0 google-chrome --ignore-certificate-errors
            https://192.168.1.100:8088/?n=$1"_"$counter >> /dev/null 2>&1 &

        #Script for capturing CPU and Memory usage for every test
        ./memCPU.sh $dest_folder $counter >> 1to1.log 2>&1 &
        memCPUPID=$!

        sleep $TIMEOUT
        echo $PASSWORD | sudo -S killall conmon >> 1to1.log 2>&1
        kill $memCPUPID
        dir_file=$1"_"$counter
        mkdir $dest_folder/$dir_file
        mv $backup_files $dest_folder/$dir_file
        (( counter++ ))
done

sleep 30
echo "Finishing test..."
echo $PASSWORD | sudo -S killall conmon >> 1to1.log 2>&1
killall v4l2_file_player >> 1to1.log 2>&1
killall chrome >> 1to1.log 2>&1
```

Listing 8: Measue and store CPU and Memory usage

```
#!/bin/bash

#Script used to measure periodically the status of the CPU and memory
PREV_TOTAL=0
PREV_IDLE=0

#Runs until the script is killed by another process
while true;
do
        CPU=(`cat /proc/stat | grep '^cpu '`) # Get the total CPU
            statistics.
        unset CPU[0]                          # Discard the "cpu" prefix.
        IDLE=${CPU[4]}                        # Get the idle CPU time.
        timeStamp=$(date +%s)
        # Calculate the total CPU time.
        TOTAL=0
        for VALUE in "${CPU[@]}"; do
                let "TOTAL=$TOTAL+$VALUE"
        done

        # Calculate the CPU usage since we last checked.
        let "DIFF_IDLE=$IDLE-$PREV_IDLE"
        let "DIFF_TOTAL=$TOTAL-$PREV_TOTAL"
        let "DIFF_USAGE=(1000*($DIFF_TOTAL-$DIFF_IDLE)/$DIFF_TOTAL+5)/10"

        # Remember the total and idle CPU times for the next check.
        PREV_TOTAL="$TOTAL"
         PREV_IDLE="$IDLE"

        #Save the amount of used memory in Mb
        total=$(free |grep Mem | awk '$3 ~ /[0-9.]+/ { print $2"" }')
        used=$(free |grep Mem | awk '$3 ~ /[0-9.]+/ { print $3"" }')
        free=$(free |grep Mem | awk '$3 ~ /[0-9.]+/ { print $4"" }')

        #Calculate the percentage
        usedmem=`expr $used \* 100 / $total`

        #Export all the data to the defined iteration in argument 2 and
            folder 1
        echo  $timeStamp"    "$DIFF_USAGE" "$usedmem"    "$total"
            "$used" "$free >> $1/log_performance_$2.txt

        # Wait before checking again one second
        sleep 1
done
```