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:8+87

Department of Communication and Networking

Professorship: Networking Technology Code: S-38

Supervisor: Prof. Jörg Ott

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

Real-time Communications over the Web (WebRTC) is being developed to be the next big improvement for rich web applications. This enabler allow developers to implement real-time data transfer between browsers by using high level Application Programing Interfaces (APIs). Running real-time applications in browsers may led to a totally new scenario regarding usability and performance. Congestion control mechanisms may influence the way this data is sent and metrics such as delay, bit rate and loss are now crucial for browsers. Some mechanisms that have been used in other technologies are implemented in those browsers to handle the internals of WebRTC adding complexity to the system but hiding it from the application developer. This new scenario requires a deep study regarding the ability of browsers to adapt to those requirements and to fulfill all the features that are enabled.

We investigate how WebRTC performs in a real environment running over an actual web application. The capacity of the internal mechanisms to adapt to the variable conditions of the path, consumption resources and rate. Taking those principles, we test a range of topologies and use cases that can be implemented with the actual version of WebRTC. Considering this scenario we divide the metrics in two categories, host and network indicators. We compare the results of those tests with the expected output based on the defined protocol in order to evaluate the ability to perform real-time media communication over the browser.

Keywords: Internet, WebRTC, Real-time Communication, Network Topologies, Performance Analysis, Congestion Control, Browsers, HTML5

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
AVPF	Audio Video Profile with Feedback
DNS	Domain Name System
DOM	Document Object Model
DoS	Denial of Service
DTLS	Datagram Transport Layer Security
FEC	Forward Error Correction
FPS	Frames per Second
HTML	HyperText Markup Language
HTTPS	Hypertext Transfer Protocol Secure
IAT	Inter-Arrival Time
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
JSON	JavaScript Object Notation
LAN	Local Area Networks
MCU	Multipoint Control Unit
MEGACO	Media Gateway Control Protocol
NAT	Network Address Translation
NR	Noise Reduction
NTPd	Network Time Protocol daemon
OWD	One-Way Delay
QoS	Quality of Service
REMB	Receiver Estimated Maximum Bitrate
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
RTT	Round-Trip Time
SCTP	System Control Transmission Protocol
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SRTP	Secure Real-time Transport Protocol
SSRC	Synchronization Source identifier
STUN	Simple Transversal Utilities for NAT

TFRC	TCP Friendly Rate Control
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 rapidly changing, the dramatical increase of Internet communication between people is forcing technology to support mobile and real-time video experiences in different ways. Applications such as *Vime* and *Skype* provide media and real-time communication over the Internet.

Technology and media are changing the way people interact with each other, communication over the internet is encouraging users to interact, talk and see content in a cost-effective and reliable way. Business structure and innovation is rapidly changing thank to the existence of reliable, low cost and simple platforms for real-time communication.

This way of communication is adding new features to services that have never been available before, they are now able to have high-engagement communication, where richer, more intimate communication is possible [5]. At the same time, this is changing traditions and habits of communication between people and transforming personal relationships [6]. 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 such as gaming or using social networks.

Furthermore, it also helps businesses to have lightweight communication alternatives that will increase their efficiency besides the size of the company.

Real-time communication is encouraging front end designers designers and device developers to turn their products into multi-functional and interoperable 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 (e.g DSL, 3G or EDGE) while affordable digital and video cameras have become integral parts in nowadays computers. Those two enablers along with the increased demand for richer applications with easy integration for the WWW are some of the reasons behind Real Time Communications for the Web (WebRTC).

WebRTC is a suite of tools that enable human communications via voice and audio where Real Time Communication (RTC) should be as natural in a web application as browsing images or visiting websites instead of requiring extra software to be installed. With this simple approach, WebRTC aims to transform something that has been traditionally complex and expensive into an open application that can be used by everybody, enabling this RTC technology in all existing web applications and giving the developers the ability to innovate and allow rich user interaction in their products using an standardized and free technology.

Many web services already use RTC technology to allow communication (e.g., Google Hangouts and Adobe RTMFP) but most of them require the user to download native apps or plugins to make it work. With WebRTC, real-time communications between users should be transparent for them, since downloading, installing and using plugins can be complex and tedious. On the other side, the usage of plugins 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 free to use and openly standardized.

1.1 Background

WebRTC is an effort to bring real-time communication APIs to JavaScript developers, allowing them to build RTC functionalities into web applications that enable different features, such as video calling platforms over the web.

Web application APIs are defined in the HyperText Markup Language (HTML) version 5 and help developers to add features to their web applications with minimal effort using JavaScript functions. APIs can be defined as a collection of methods and callbacks that help developers to access available technologies in the browser, they are used in web development to access the full potential of browsers and compute some part of the dynamic web applications on the client side.

WebRTC APIs works by combining two different technologies, HTML and JavaScript, HTML is the de facto markup 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 formed between different API that integrate with each other to provide flexible RTC in the browser. However, the final goal is to allow developers to create plugin-free real-time web applications that are cross compatible with different browser vendors and operating systems.

The flexibly of WebRTC allow many uses of RTC thus foreseeing many new topologies that will emerge. Developing RTC technologies over the Internet has been always closely related to the topology distribution of the nodes that are being connected, performance issues in different topologies usually restrict the amount of nodes available for the session. From the simple point-to-point topology to the mesh composition, choosing the right option for each application is very important to deliver a great user experience.

1.2 History

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 through different versions to increase its usability thanks to the feedback given by web developers.

The first W3C announcement of WebRTC was done in a working group in May 2011 [7], and the official mailing list started in April 2011 [8]. During the first stage of discussion, the main goal was to define a public draft for the version 1 of the API implementation and a timeline with the goal to release the first final version of WebRTC. The W3C public draft of WebRTC was published on 27th of October 2011 [9]. This first W3C draft only specifies how to send media (audio and video) over the network to other peers, it defined the way browsers would be able to access media devices without the use of any plugin or external software.

WebRTC project got involved with the IETF in May 2011 [10]. The initial milestones of the IETF initially marked December 2011 as the deadline to provide the information and elements required for the W3C to design the first version of the API. On the other side, the main goals of the IETF working group are the definition of the communication model, session management, security, NAT traversal solution, media formats, codec agreement and data transport [11]. In June 2011 Google publicly released the source code of their WebRTC API implementation [12].

WebRTC APIs are integrated within the browser and accessible using JavaScript in conjunction with the Document Object Model (DOM) interfaces. Some of the APIs that have been developed for WebRTC are not part of the HTML5 W3C specification but are included into the Web HyperText Application Technology Working Group (WHATWG) HTML specification.

1.3 Challenges

WebRTC is a suite of protocols that share the available device resources with many other applications. Due to the little experience in WebRTC environments sharing the available resources, we find some lack of documentation or previous literature regarding congestion analysis compared with other technologies.

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

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 that could change in future versions of WebRTC, even dough it should not affect the overall conclusions.

Considering the fact that WebRTC is still 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 issues could have been solved.

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

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

On the other side, we will also study the available topologies for RTC and which one of them fit in a WebRTC environment, trying to understand the limits of the protocol in different scenarios and topologies. All the structures that can be implemented using WebRTC might have some restrictions in performance and user experience that we need to study and understand, those limitations are important when designing WebRTC applications.

1.4 Contribution

Investigate how WebRTC performs in a real environment trying to evaluate the best way to set multiple peer connections that handle 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.

By using metrics related to RTC protocols we expect to understand the way WebRTC performs when handling in different environments.

1.5 Goals

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

- WebRTC performance in different topologies and environments using real sources of video and audio that are 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 is 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 covers an overall analysis and usage experience of WebRTC, providing some valuable feedback for further modifications on the existing API.

1.6 Structure

This thesis is structured as follows:

1. Introduction of real-time communication
2. Description of different protocols used for real-time communication and the APIs that are built within WebRTC
3. Definition of different possible topologies used for real-time communication
4. Analysis of the required metrics used to evaluate WebRTC performance
5. Environment setup used to evaluate WebRTC

6. Result and analysis of the different scenarios that can be given in WebRTC applications

7. Conclusions of the thesis

2 Real-time Communication

Real-time Communication is defined as any method of communication where users can exchange data packets (e.g media, text, etc.) with low latency in both direction. The purpose of RTC is widely seen as a way to communicate between people. This is done in a two-way scenario where both users are senders and receivers of media packets, live video is a one-way configuration with one unique source of data and one or multiple receivers. In the first RTC configuration, latency is very important in order to achieve good quality for bidirectional communication between both users whereas the live scenario can tolerate some latency in the link. In two-way communication data can be transmitted using multiple topologies, they are either peer-to-peer or using a centralized relay.

Some other ways of transmitting data include multicast or broadcast. In the development of this thesis we do not study multicast and broadcast streaming.

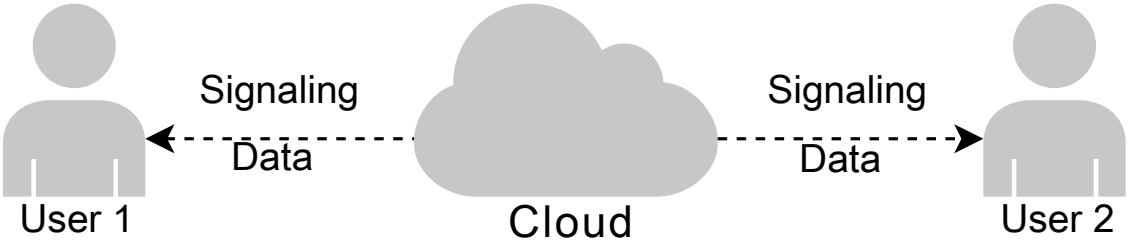


Figure 1: Real time communication between two users over the Internet.

Figure 1 shows an RTC scenario for two users, the technology providing the communication may differ in each situation but the goal is always the same.

RTC has a characteristic that is 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 procedure is used by the protocols to check the capabilities of the two entities before proceeding to send the media. The signaling channel is used to negotiate the codec agreement and NAT traversal methods that are used 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 starts to flow to the receiver, this stream may include media (audio or video) and different types of data (e.g binary, text, etc.). During the transmission we may require also some extra signaling messages to be exchanged in order to maintain the path or adapt the constraints to the actual network conditions, at the same time, features might change during the session.

2.1 Session Initiation Protocol (SIP)

SIP is a protocol used to create, modify and terminate multimedia sessions. This protocol features real-time communication between different peers with multiple

optional extensions, those extensions allow the usage of instant messages or subscriptions to different events. SIP final Request for Comments (RFC) was published in June 2004, this document describes the original functionalities and mechanisms of SIP [1].

Other features of SIP is the ability to invite participants to already existing sessions in order to build multicast conferences. SIP also gives support for name redirection being a federated protocol regardless of the user network location.

The process of SIP includes the user location, availability, media capabilities, setup of the session and management of itself. On the other side, SIP can also be defined a suite of tools that are built together with other existing protocols such as Real-time Transport Protocol (RTP) , Real-time Transport Streaming Protocol (RTSP) , Session Description Protocol (SDP) and Media Gateway Control Protocol (MEGACO) .

SIP provides low level services to deliver messages between users, for example, it could deliver SDP messages to be negotiated between the endpoints in order to agree on the parameters of a session.

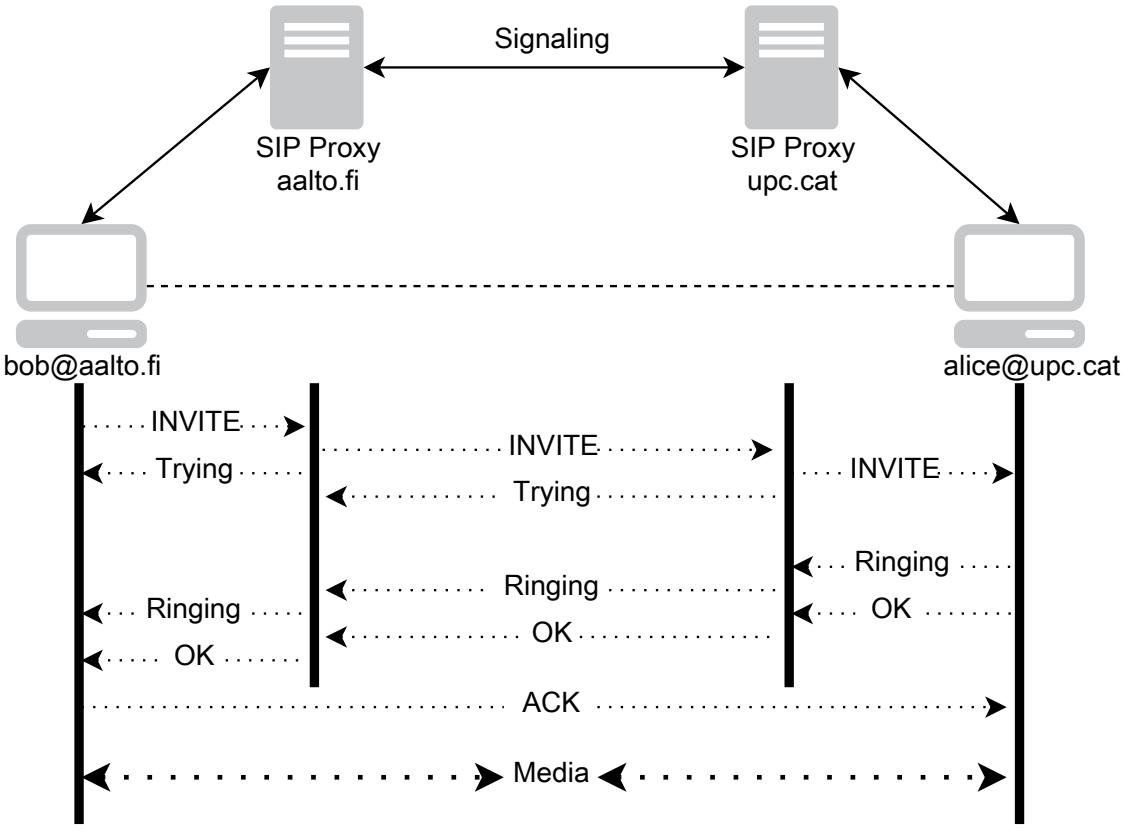


Figure 2: SIP session establishment example. Source [1].

Figure 2 shows a typical example of a SIP scenario with a message exchange between two endpoints. SIP can use SDP Offer/Answer model for the session description or capabilities negotiation between the end-points [13] and RTP for the media transport.

On the other side, SIP also relies on some elements called proxy servers that help to route requests to the final destination. Those proxies are represented in Figure 2.

SIP uses a wide range of methods that help the protocol to understand the type of message that is exchanging between peers. Figure 2 shows an example of message exchange between endpoints. INVITE request is an indicator to the receiver that someone is trying to contact, Trying indicates that the INVITE has been received and the proxy is trying to find the correct path to the destination, Ringing message represents hold for answer from the other peer. Finally, once the receiver chooses to answer a OK message is generated to indicate that the media session can start.

The different media parameters are negotiated using the SDP bodies transmitted into the SIP methods previously mentioned. Those parameters are agreed between the peers in order to provide compatibility.

This protocol has been used for some time and improved due to many iterations and additions to itself. The knowledge raised from this technology helped to have a better understanding of the requirements when building real-time media protocols such as WebRTC.

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 deliver multimedia and RTC between users.

Adobe Flash is a media 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 enable this content we need to install Adobe Flash plugin on the computer.

Adobe uses a proprietary programming language called JavaScript Flash Language (JSFL) and ActionScript. 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. Flash Player is available for most platforms, except iOS devices, and is present in about 98% of all Internet-enabled browser devices. This plugin allows developers to access media streams using external devices such as cameras and microphones to be used along with RTMFP.

This protocol is implemented by using Flash Player, Adobe Integrated Runtime (AIR) and Adobe Media Server (AMS) [14].

RTMFP uses Adobe Flash to provide media and data transfer between two end points over UDP [14]. RTMFP requires a plugin to be installed in order to be functional being a proprietary protocol. It also handles congestion control over the path and NAT transversal issues. One of the biggest differences is that, compared to SIP, RTMFP does not provide inter-domain connectivity and both peers must be in the same working domain to be able to communicate.

Media transfer is encrypted, this feature is provided in RTMFP by using proprietary algorithms and different encryption methods. RTMFP architecture allows

reconnection in case of connectivity issues and works multiplexing different media streams over the session. On the signaling part, Adobe uses an application server called Cirrus [15] (Figure 3) to handle the signaling between the different participants of a session. This service provides support to handle different topologies such as: end-to-end, many-to-many and multicast. Those structures rely in the use of overlay techniques in multicast and mesh scenarios.

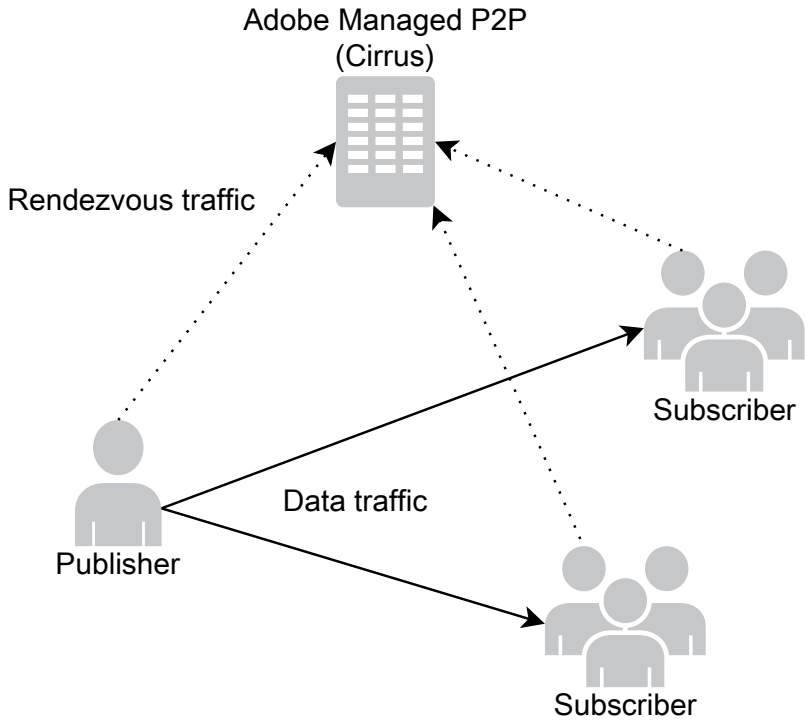


Figure 3: RTMFP architecture using Cirrus.

Some of the most valuable features is the possibility to integrate P2P multicast topologies where one source sends a video to a group of receivers.

2.3 Web Real-Time Communication (WebRTC)

WebRTC is part of the HTML5 proposal, it is defined in the W3C [7] [3], and enables RTC capabilities between Internet browsers using simple JavaScript APIs, providing video, audio and data P2P without the need of plugins. This API is in the process of replacing the need for installing a plugin to enable P2P communications between browsers, WebRTC uses existing standardized protocols to perform RTC.

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 include some WebRTC APIs are: Google Chrome, Mozilla Firefox and Opera. Other providers, such as Internet Explorer, are in process of building prototypes for WebRTC [16].

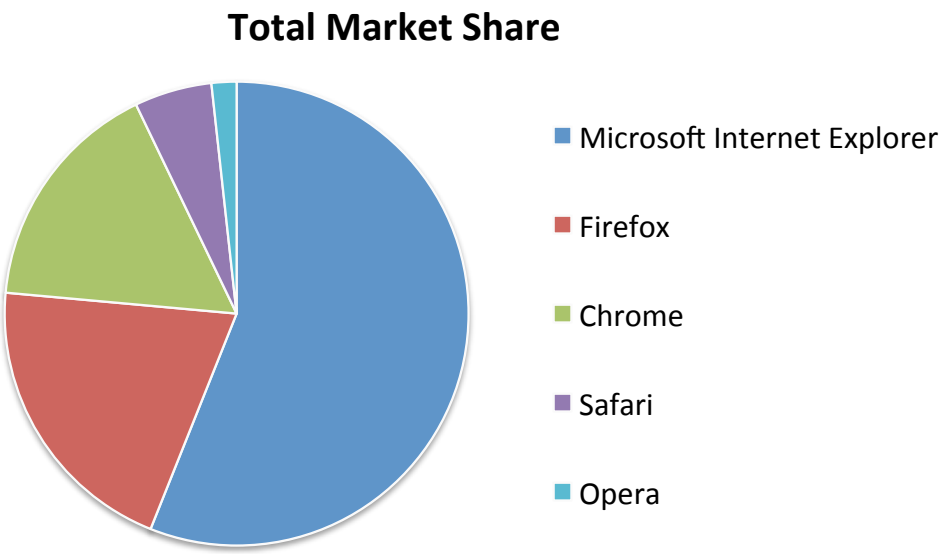


Figure 4: Market share of browser vendors by April 2013 [2].

With WebRTC, developers can provide applications for most of the desktop devices available, mobile devices will integrate WebRTC as part of their HTML5 package to also enable RTC soon [17].

WebRTC is composed of two important APIs that enable real-time features, GetUserMedia and PeerConnection. Both of them are accessible by JavaScript on the browser.

2.3.1 GetUserMedia API

WebRTC applications use the GetUserMedia API to allow access to media streams from local devices (video cameras and microphones).

This API itself does not provide RTC, but provides the media to be used as simple HTML elements in any web application.

GetUserMedia API allows developers to access local media devices using JavaScript code and generates media streams to be used either with the rest of the *PeerConnection* API or with the HTML5 video element for playback purposes [3]. *GetUserMedia* is already interoperable between Google Chrome, Firefox and Opera [18].

GetUserMedia proposal removes the need of using Adobe Flash to access the media device and also the plugin requirement.

Figure 5 illustrates how the browser access the media and delivers the output to JavaScript. We use the *GetUserMedia* API to build WebRTC-enabled applications for RTC video conferencing. The video tag is an HTML5 Document Object Model (DOM) element that reproduces local and remote media streams.

In Figure 5 *MediaStream* is the object returned by the *GetUserMedia* API methods, this object is contains *MediaStreamTracks* that carry the actual video and audio media. The goal of using this architecture is to be able, in the near future, to include multiple sources of video and audio multiplexed over the same stream from differ-

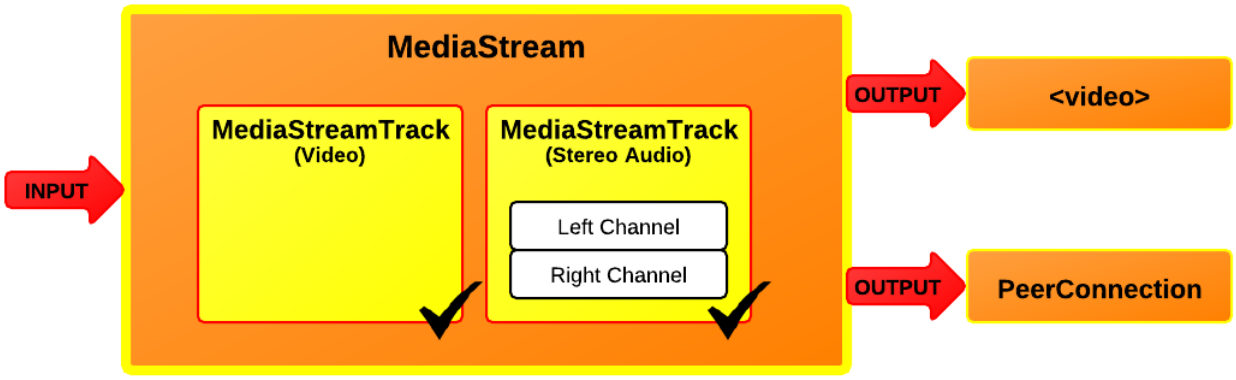


Figure 5: Media Stream object description. Source [3].

ent devices. Different alternatives include the implementation of overlay topologies by forwarding media from one peer to the other by including different *MediaStreamTracks* into the same *MediaStream*. Furthermore, *MediaStreams* handle the synchronization between all the *MediaStreamTracks* included for proper playback at the application level, by this it assures that audio and video will be always synchronized.

GetUserMedia API works using a JavaScript fallback method, this method returns a *MediaStream* object to the application that is played in the HTML web application or used in the *PeerConnection* API. A sample example of this method can be seen in Listing 1.

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

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

function gotStream(stream) {
        //Stream is the MediaStream object returned by the API
        console.log("GetUserMedia succeeded");
        document.getElementById("local-video").src =
            createObjectURL(stream);
}
```

In Listing 1, we are using the video and audio media from our devices to be played in an video HTML element identified as *local-video*.

GetUserMedia API also allow developers to set some specific constraints for the media acquisition. This helps applications to better adapt the stream to their requirements, those *cameraConstraints()* are provided by a JavaScript Object Notation (JSON) library.

2.3.2 PeerConnection API

WebRTC uses a separate API to provide the networking support to transfer media and data to the other peers, this API is named *PeerConnection* [19]. PeerConnection API bundles all the internal mechanisms of the browser that enable media and data transfer, at the same time it also handles all the exchange signaling messages with specific JavaScript methods.

Figure 6 describes the topology used in WebRTC for a bi-directional media session, with the messages being sent either by WebSockets or by HTTP long polling. Messages are built using a modified bundled version of SDP, WebRTC signaling messages are similar to SIP as they use SDP bodies for the agreement.

SDP is widely used in SIP to provide media and NAT reversal negotiation between two different endpoints prior to establish data transmission. This protocol is used in WebRTC in a modified version that allow the usage of multiple media descriptions over a single set of Interactive Connectivity Establishment (ICE) . Usually, in other conditions, different media types will be described using different media descriptions.

This new feature is described as Bundle [20] and can be used along with the existing SDP Offer/Answer mechanism to negotiate the different media ("m=" lines) on the session.

By using Bundled SDP, WebRTC multiplexes all the traffic using a single port, this means that media, data and monitoring messages are sent over the same port from peer to peer, traffic is sent over UDP or TCP [21]. PeerConnection API provides signaling and NAT transversal techniques, this part is very important to guarantee a high degree of success when establishing calls in different environments.

PeerConnection P2P session establishment system works in a constrained environment designed to provide some degree of legacy for other SDP based technologies. Figure 6 shows how a WebRTC simple P2P scenario works, the server used for signaling is a web server. WebRTC scenarios do not work easily in a federated environment such as SIP.

On the other side, signaling is not standardized in WebRTC and has to be provided in the application level by the developer.

Figure 6 does not show relay servers that provide NAT transversal solutions described in Section 2.3.5 . When developing a WebRTC application, those servers must be provided into the WebRTC *PeerConnection* configuration when starting a new call as seen in Listing **??**.

Listing 2: Simple example of *PeerConnection* using JavaScript

```
//XXXX represents the stun server address
var pc_config = {"iceServers": [{"url": "stun:XXXX"}]};
pc = new webkitRTCPeerConnection(pc_config);
pc.onicecandidate = iceCallback1;

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

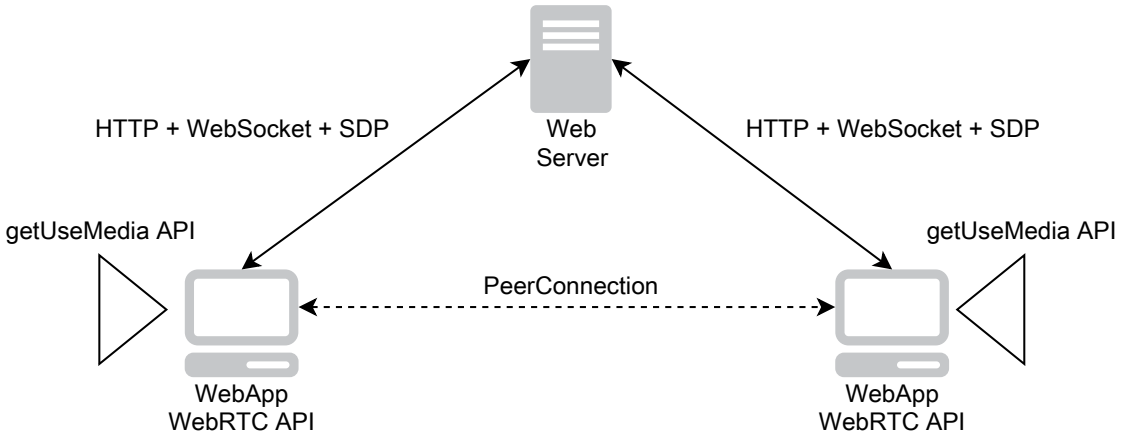


Figure 6: WebRTC simple topology for P2P communication.

```
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);
}
```

Listing 2 represents 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 stun configuration for the NAT transversal process: *var pc_config = {"iceServers": [{"url": "stun:XXXX"}]};*.

2.3.3 Control and Monitoring

Control and monitoring is an important part of all RTC protocols, this part is usually handled by the JavaScript API.

Media constraints are defined as a set of parameters that limit the media quality when processed from the devices such as microphone or webcam, those parameters are usually related to video size or frame rate in WebRTC. However, in WebRTC we can also adapt the maximum link rate through an special set of constraints.

Those parameters are implemented through the Statistics Model and Constraints defined in the W3C draft [19], these set of methods are part of the actual *PeerConnection* API defined in section 2.3.2. Once the *PeerConnection* is 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 form the remote side. We focus on those remote stats to study the status of the path and to obtain the desired metrics for monitoring.

To access the statistical data retrieved from the control messages we need to use the *getStats()* method of the *PeerConnection* object defined in the draftt [19], this method allow the application to access that data in a JSON format that might require some post-processing. Statistical models are useful for the developers to monitor the status of their WebRTC applications and adjust the attributes of the *PeerConnection*.

Within constraints, developers are able to change media capture configuration by setting parameters such as 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 in Pixels for Width and Height. Frames per Second in
    minFrameRate
var constraints = {
        "audio": true,
        "video": {
                "mandatory": {
                        "minWidth": "300",
                        "maxWidth": "640",
                        "minHeight": "200",
                        "maxHeight": "480",
                        "minFrameRate": "30"
                },
        "optional": []
        }
}

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

```
}
```

Both constraints objects are added to the *GetUserMedia* and *PeerConnection* methods when building the new session. 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 JavaScript developers.

One of the difficult parts in the standardization process has been to decide about the complexity level of the API, how much is available to be accessed by the developers and which configurations or mechanisms should be automated in the browser. After long discussion, WebRTC is now using Javascript Session Establishment Protocol (JSEP) [22], this API is a low level API that gives the developers control of the signaling plane allowing each application to be used in specific environments.

The media processing is done in the browser internals but most of the signaling is handled in the JavaScript plane by using JSEP methods and functions. Figure 7 shows 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.

Furthermore, JSEP also handles the state management of the session by building the specific SDP message that is forwarded to the other peer. NAT traversal mechanisms are activated in JSEP also, those mechanisms are described in chapter 2.3.5.

Another 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 technique 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 [22].

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 build 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 is that there are the multiple possibilities to adapt WebRTC to the requirements of each specific application disregard its advantages a disadvantages without being sensible to pick an specific design at a time.

2.3.5 Internals of WebRTC

WebRTC has multiple internal mechanisms that enable the RTC in the browser level by using APIs. Those mechanisms work together to accomplish all the goals

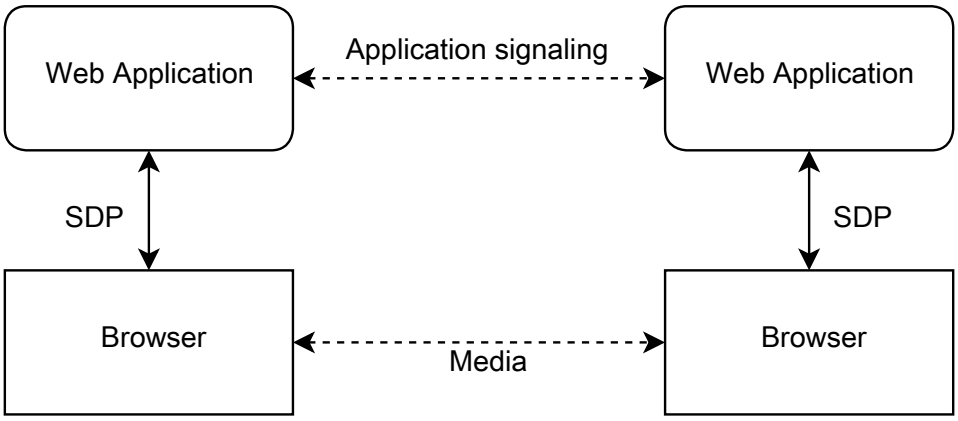


Figure 7: JSEP signaling model.

of WebRTC features, some of them are related to the network level and others to video access.

One of WebRTC main issues is NAT transversal difficulties, this problem usually affects all RTC related technologies. Interactive Connectivity Establishment (ICE) is a technique that helps WebRTC to decide which is the best way to bypass NATs and firewalls, ICE is widely used in media communications and has proven to be reliable when choosing the best option to enable connectivity in restrictive environments [23].The enablers that work together with ICE for Real-time protocols are Simple Transversal Utilities for NAT (STUN) and Traversal Using Relays around NAT (TURN) [24] [25].

TURN and STUN servers 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. STUN server function is to discover the available IP addresses and ports that allow direct connectivity to a target machine placed behind a firewall or NAT, those interfaces are named *candidates*, this information is provided to the sender that process it in order 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 connection to no other *candidate* was established. TURNs work by rerouting the traffic from one peer to the other.

All traffic in WebRTC is done over UDP and multiplexed over the same port. In case of TURN the traffic can be sent over TCP also.

Media encoding in WebRTC is done through codecs implemented inside the browser. Mandatory-to-Implement codecs for audio are G.711 and Opus. G.711 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 G.711, 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 [26]. Opus can be easily adjusted for high and low encoding rates, applications can use additional codecs.

Along with the codecs, the audio engine for WebRTC also includes some features such as Acoustic Echo Cancellation (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 (local loop), with this, WebRTC solves the issue of the 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. When both mechanisms are working properly, the amount of rate required by the audio channel is reduced as the unnecessary noise is removed form the spectrum. AEC and NR mechanisms provide a smooth audio input for WebRTC protocol.

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 as *Data Channel* allows generic data exchange in a bidirectional way between two peers [27]. Non-media data in WebRTC is transferred using System Control Transmission Protocol (SCTP) encapsulated over Datagram Transport Layer Security (DTLS) [28] [29] [27].

The encapsulation of SCTP over DTLS on top of ICE/UDP provides a NAT traversal solution for data transfer that combines confidentiality, source authentication and integrity. 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.

WebRTC provides Secure Real-time Transport Protocol (SRTP) to allow media to be secured.The key-management for SRTP is provided by DTLS-SRTP which is an in-band keying and security parameter negotiation mechanism [4]. Figure 8 illustrates the full protocol stack for WebRTC described in this chapter.

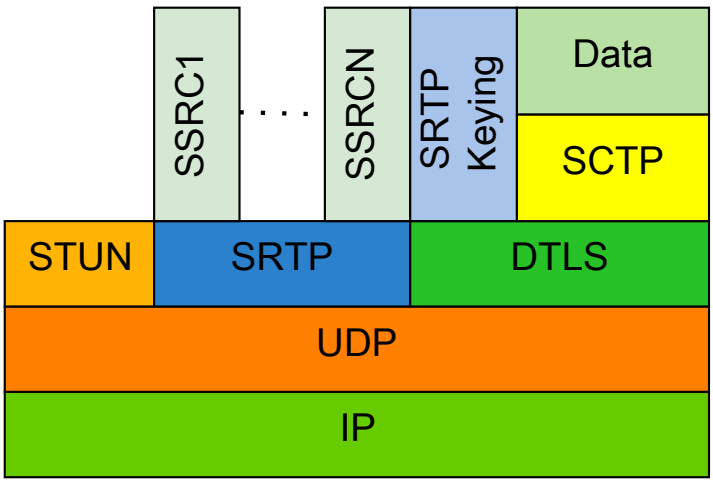


Figure 8: WebRTC protocol stack. Source [4].

Quality of Service (QoS) for WebRTC is also being discussed in the IETF and a draft is available with some proposals [30]. 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 is originated on the Internet Service

Providers (ISPs) as they might be using their own packet marking with different DiffServ code-points, those packets are not interoperable between ISPs, there is an ongoing proposal to solve this problem by building consistent code-points [31]. Otherwise, clients might also be sending too much data for the specified path reducing the effectiveness ofDiffServ. Each specific application will mark Audio/video packets with the designed priority using DSCP mappings [30].

Officially, there is no congestion control mechanisms for WebRTC, the only mechanism actively used is circuit breakers for RTP [32].

Furthermore, Chrome specifically uses a Google congestion control algorithm that enables congestion control mechanisms for rate adaptation [33]. The aim of this algorithm is to provide performance and bandwidth sharing with other ongoing conferences and applications that share the same link. This algorithm is defined in Section 6.

2.3.6 Security concerns

To handle the signaling process WebRTC uses a web application server, peers exchange messages with each other through the web server in multiple different ways. By using this system WebRTC provides high flexibility for developers to allow multiple scenarios, on the other side, it also has some important security concerns [34]. Figure 6 presents a simple topology for a WebRTC call, the web application server handles the signaling messages to the peers and the media transport is done between them and provided by the browser.

Obviously, this system poses a range of new security and privacy challenges different from traditional VoIP systems. Considering that WebRTC APIs are able to bypass Firewalls and NAT, Denial of Services (DoS) attacks can also become a threat. On the other side, malicious JavaScripts could also perform calling to unknown devices.

Browsers execute JavaScript scripts provided by the web applications, this may include malicious scripts, that in the case of WebRTC could lead to some privacy issues. 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 could execute a variety of actions in that browser. At a minimum, it should not be possible for arbitrary sites to initiate calls to arbitrary locations without user apprehension [35]. To approach this issue, 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 [36]. Those situations are given when a third-party server provide JavaScript scripts to a different domain to be one accessed. This script cannot be trusted by the original accessed domain as it could trigger browser actions that might harm the privacy. For example, in WebRTC, the user could load a malicious script from a third-party entity in order to automatically build a WebRTC call to an undesired receiver without the user noticing this situation. 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 include 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 could build one *PeerConnection* to the receiver and an extra one to a remote peer that could store the call without the user noticing this behavior. Those problems are not only related to WebRTC and tend to happen in related protocols.

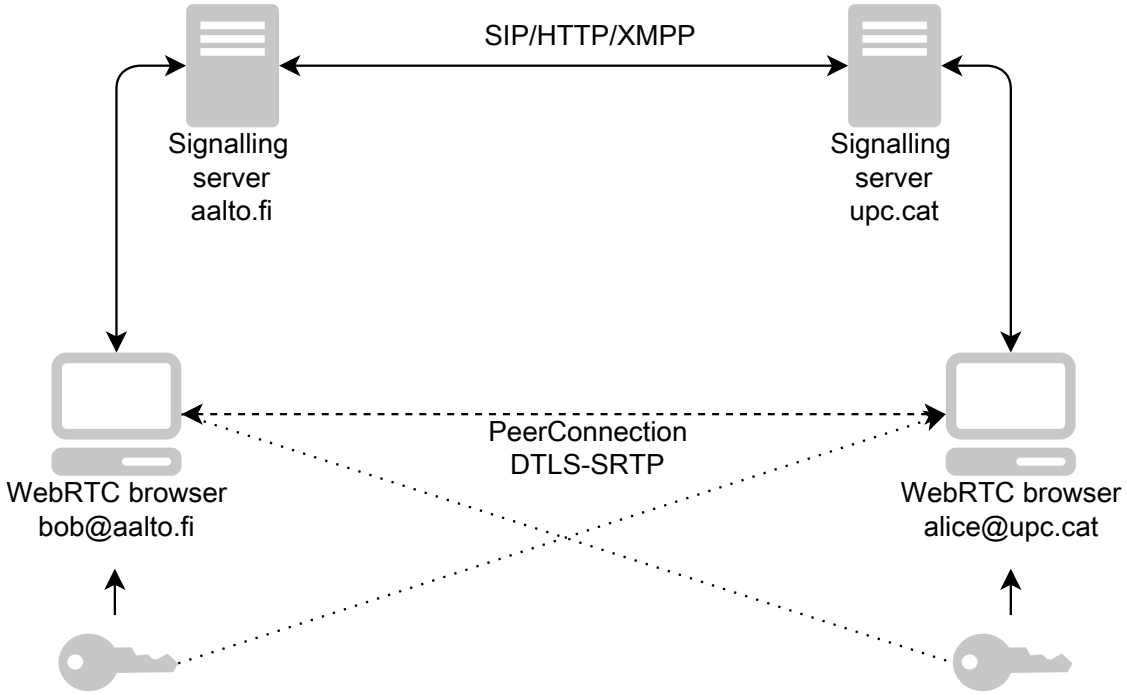


Figure 9: WebRTC cross-domain call with Identity Provider authentication.

Calling procedure is done using the JavaScript provided by the server, this may be a problem as the user must trust an unknown authority provider. WebRTC calling services usually rely on Hypertext Transfer Protocol Secure (HTTPS) for authentication where the origin can be verified and users are verified cryptographically (DTLS-SRTP). Browser peers should be authorized before starting the media flow, this can be done by the *PeerConnection* itself using some Identity Provider (IdP) that supports OpenID or BrowserID to demonstrate their identity [37]. 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, 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 secure certificate and has been authorized access to the media, *GetUserMedia* access becomes automatic after the first time under the same domain, otherwise, the user has to verify the access for each call. Once the media is acquired, the 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 sourced real time protocol. Some use cases for WebRTC also incorporate some level of vulnerability as the JavaScript is going to be provided by a third-party, in the use case of media streaming, advertisement or call centers where service providers could pick data form the users and store them for further usage [34].

2.4 Comparison between SIP, RTMFP and WebRTC

After describing various RTC technologies and two important 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.

	SIP	RTMFP	WebRTC
Plugin-enabled	No	Yes	No
Cross-domain	Yes	No	Maybe
Audio	Yes	Yes	Yes
Video	Yes	Yes	Yes
Data	Yes	No	Yes
NAT Traversal	Yes	Yes	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	Yes	No	Yes
SCTP	Yes	No	Yes

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

RTFMP is a proprietary protocol which mean that might have its own mechanisms other than the standardized ones stated on Table 1 to solve some of the issues.

All the protocols explained in this section are designed to provide similar real time features but in different ways, meanwhile SIP is a protocol that helped to develop some of the important technologies, such as RTP and SRTP, that are used in other technologies, is still not easily accessible for web developers. On the other side, RTMFP provides a licensed alternative for real time communication having some mechanisms not standardized and with compatibility issues between devices.

From the mobile perspective, SIP is used in mobile technology and WebRTC has announced to be compatible with future versions of iOS and Android [17]. 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 for NAT traversal that is not standardized, SIP and WebRTC use a conjunction of TURN, STUN and ICE mechanisms.

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

2.5 Summary

We can conclude that real-time media protocols have been developed for quite a long period. However, an standardized, open source solution has not been provided yet. WebRTC and SIP share a basement of principles that work together, we could state that WebRTC could not exist as it is now without the previous knowledge provided by SIP. Besides, the usability of SIP in web applications is still very complex and prohibitive for most web developers.

Furthermore, RTMFP has shown more ubiquitous availability on user's devices than any specific web browser. The main problem with RTMFP approach is that the protocol for end-to-end media path is proprietary, so interoperating with existing VoIP solutions can be inefficient and developers rely on vendors plugins to take care of any platform incompatibilities.

WebRTC solution may not be perfect but is a good start to provide interoperable real-time solution for web applications.

3 Topologies for real-time multimedia communication

In this chapter we discuss different possible topologies that can be used along in real time media communication.

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 configurations. Topologies enable different RTC architectures with optimal performance for each specific scenario.

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

The usage of NAT traversal mechanisms in WebRTC is crucial and at the same time it increases the complexity of the browser internals. STUN and TURN servers must be reachable from the browser perspective in order to provide the ports and IP alternatives to connect, those are gathered into *candidates* that are evaluated by the ICE on the browser. ICE proceeds with the best option to perform the connection.

All the previous mechanisms are supposed to provide high level of success probability but might fail in very restrictive environments. To solve some of the issues, WebRTC allows UDP and TCP (using TURN) packet transport, this is done to enable connectivity even in very restrictive environments that could have UDP packet drop mechanisms.

For some topologies that include the establishment of multiple *PeerConnections* resource usage can be a big problem (e.g. mesh, one-to-many or tree). Considering that system capacity relies in how the OS architecture handles processes, 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 other architectures this might be handled different. Media encoding usually consume most of those resources becoming a bottleneck for some scenarios.

3.1 Point-to-Point

The simplest topology is a permanent session 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 cross-domain communication alternatives such as SIP. Scenarios such as Figure 2 are difficult to design in a WebRTC application, on the other side, Figure 6 represents the most common WebRTC point-to-point scenario.

With point-to-point topology we can have traditional dedicated paths where the resources are reserved for each call. In small Local Area Networks (LAN) we use dedicated paths between two WebRTC users, this path can go through the switch or relay but it is unlikely that is going to change the routing. For WebRTC calls over

the public internet, the route can change at any time trying to use the optimal path, this is done in packet-switching technologies where the route is set up dynamically. However, the user perception is that the communication is done end-to-end without noticing any change on the network nodes.

From usability perspective, different environments might require point-to-point topologies, direct calls between two users or real time communication for IM can be possible scenarios.

3.2 One-to-Many

One-to-many or star topologies are one of the most common network topologies for media streaming (e.g Windows Media Server or RTMFP), this kind of topology consist on a central node that transmits streams to the rest of nodes connected to it. In the WebRTC example of Figure 10, the central node might be also receiving real time data in difference of the traditional streaming scenarios providing P2P communication between the peers and the central node.

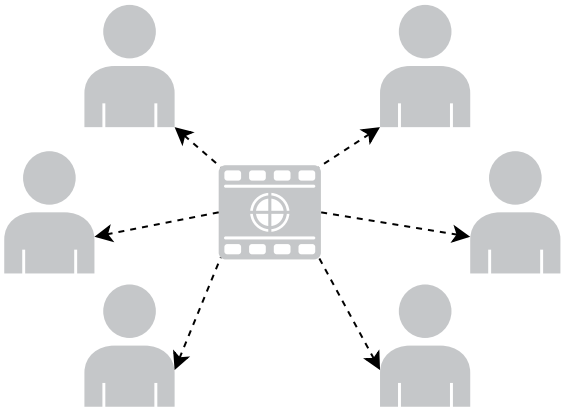


Figure 10: One-to-many topology for real time media.

Star scenarios are known as a type of multicast, one source sends the media to the different clients that connect to the origin. When using this topology, the common uses are related with video and audio streaming to multiple peers, TV media and streaming conferences are popular use cases.

Live streaming is a common in-demand scenario for internet TV channels, this topology adapts to the requirements of the providers as subscribers can join the suggested channel when desired.

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

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 with an HMTL5 web application. Music bands also

could take advantage of this scenario by being able to transmit his show to the audience with feedback in real time or having the members playing from different geographic areas. All the previous examples 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 star topology we have a video, audio and data streaming connection from one source to multiple devices. This might cause a huge load on the source when having multiple *PeerConnection* running, central node performance can be a big constraint in this scenario. Observing other topologies, in most cases, media delay on the network is not as important as other options due to the one-way communication only. In most scenarios, video and audio is not required to be received on the source, so having the media delayed a couple of seconds is not going to affect the user experience in the call. Those scenarios are one-way only use cases.

From the client perspective, the *PeerConnection* stablished is easy to handle as in most cases no data is going to be sent back to the source, except the RTCP control messages.

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. Conferencing systems are widely extended 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 increasing 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 shall be dependent on the amount of people in the conference. The number of *PeerConnections* can grow rapidly based on 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 is a device used to bridge streams in conferences, it multiplexes, mixes and encodes media of different sources to be sent over one gateway. MCU usage could be a good alternative when designing WebRTC infrastructures such as video conferencing, the ability to multiplex different streams into the same channel is going to directly affect on how the client performs when reproducing the video.

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

MCUs receive the streams from the clients and multiplex them over one unique channel, this provides good scalability from the client perspective because it is only building one connection even there are multiple peers on the conference.

Some MCUs may have to encode and decode media on the fly, this is can be difficult in real time applications but can provide different encoding options to adapt the stream output to the link conditions. Usually transcoding is not suitable for real time environments.

Drawbacks on the MCU model affect the dependency of the end nodes from the MCU, if the MCU fails to give good latency and performance, the call quality is affected and receivers do not get the expected response. Load in the MCU can be very high when multiple conferences are being stablished, this requires abundant resources and good throughput.

Point-to-point topologies do not require much resources from the service provider, but for the MCU scenario the service provider has to be able to scale properly.

3.5 Overlay

Overlaying media streams is the ability of a peer to forward media to a third party. Topologies that use overlay 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-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.

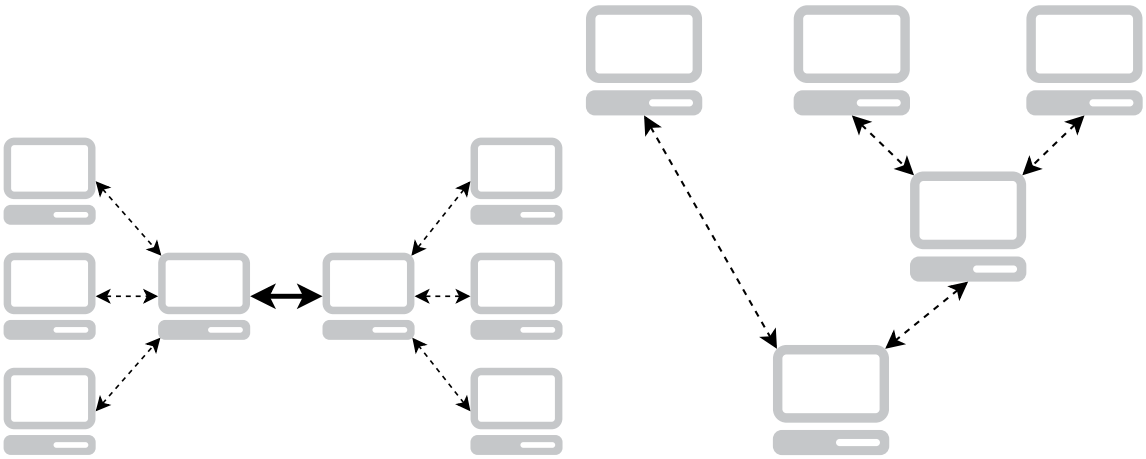
WebRTC does not provide native support for media overlay yet, but it is planned to implement those features in future versions of the API. Traditionally overlay has also been used for media streaming over the internet.

3.5.1 Hub-spoke

Hub-spoke 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 for some scenarios as it requires less connections to perform a full mesh communication in the network.

This is a centralized model, we might have problems if the key nodes of the topology fail. 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 system is able to select the best candidate where the rest of nodes connect to. When this



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

Figure 11: Overlay topologies.

happens that node is handling and forwarding more data that in a standalone call, sometimes without knowledge.

This topology uses the concept of overlay previously described. Hub-spoke environments are also used for logistics in the world, for delivering products and goods around the globe, focusing in bridges over the continents, goods in Europe are distributed within an 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 number 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 can have connectivity problems if a node fails to keep the link up, all the layers under that node are going to 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, large tree topologies won't be the best candidates given the delay produced 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 remote areas.

4 Performance Metrics for WebRTC

This section describes metrics to monitor the performance of WebRTC topologies, WebRTC environments require a specific approach and metrics to define how the protocol behaves in different scenarios.

Some issues affect how WebRTC performs, these range from the resources available in the peers to the state of the link. In the following chapters we describe some of them that we use in our study cases.

Other different issues come with the ability of WebRTC to share link capacity with other ongoing sessions or different traffic when having parallel calls. This might be a problem for the quality of the session and one reason for call failure.

There are two different type of metrics, from one side metrics related to network performance and on the other side those to the host. Some of them are going to be similar to rate and congestion but others might be more close to the performance of the actual API implementation on the browser.

4.1 Simple Feedback Loop

Traditionally, real time media communication always rely on inelastic traffic, this type of traffic has low tolerance to error as packets should always arrive in time for playback, this is more important than having 100% packet delivery rate. Elastic traffic is common in applications that do not require real time data to be sent and is tolerant to delay, having better data delivery performance compared to inelastic traffic.

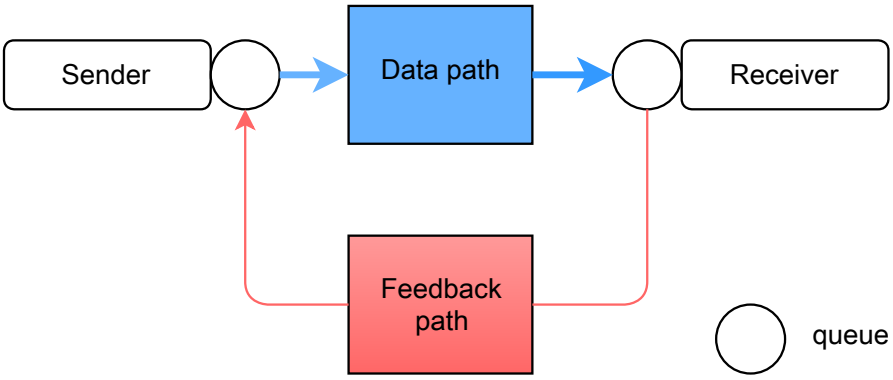


Figure 12: Multimedia feedback loop.

Figure 12 shows a simplified feedback model for multimedia communication, the feedback path carries messages with the link characteristics that help the sender to adjust its congestion mechanisms. Those feedback messages are sent periodically by the receiver and are very important to repost the status of the link.

4.2 Network metrics

Metrics defined in this chapter are only related to those provided by the network, those metrics usually trigger the congestion mechanisms in WebRTC changing the behavior of the stream according to the constraints on the link [38] [33].

With real time media applications, congestion directly affects in traffic delivery rate and performance, if the media generation rate is lower than the available channel capacity there is no need for rate adaption. However, losses and available capacity of the link can vary on time requiring adaptation from the sender side to the new constraints. This adaptation is done by analyzing the receiver feedback packets sent to the sender using the simple feedback loop.

Three global factors are considered when analyzing network links: *loss*, *bandwidth* and *delay*.

4.2.1 Losses

Loss rate indicate packet losses during the transmission over the path. Usually packet losses affect directly the quality of a call. In WebRTC, packet loss is a direct indicator of the media quality of the ongoing WebRTC transmission.

When monitoring WebRTC calls we use packet loss as an indicator to adapt the media constraints, we can use the output provided by the packet loss to reduce the frames per second that *GetUserMedia* has configured to obtain the media.

Losses can also trigger some specific JavaScript mechanism that monitors the call quality and analyzes the overall ongoing session, this mechanism could average some other metrics in order to give an output value of the media quality. Given this value, the constraints regarding video size, frames per second and bandwidth limit can be configured into the *GetUserMedia* and *PeerConnection*ÊAPIs for better adaptation to the path conditions.

WebRTC uses RTCP packets for control and monitoring on the ongoing stream [39]. In RTCP losses are reported in the feedback message, this metric does not include discarded packets by the protocol.

Losses are calculated in a period of time, with this we shall be able to see how much loss rate we have in a certain path.

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

Equation 2 calculates the estimated packet loss we have on the link. This operation is done periodically by our statistical API.

4.2.2 Round-Trip Time (RTT) and One-Way Delay (OWD)

Delay in a link can be measured form different perspectives, One-Way Delay (OWD) indicates the time it takes for a packet to move from one peer to the other peer, this time includes different delays that are produced along the link. OWD is calculated form the time taken to process the packet in both sides (building and encoding),

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.

$$OWD = delay_{propagation} + delay_{queues} + delay_{serialization} + delay_{processing} \quad (3)$$

Considering the structure of WebRTC, one of the most important delays that we have to measure and study is the processing delay, as our applications are executed in a multiple layer structure. Running on top of the browser can affect the performance compared to other technologies that run directly over the OS. Delays in our case are symmetric as we are continuously sending and receiving media, low delay is important in order to reproduce the streams in the best quality possible and avoid uncomfortable communication.

OWD and RTT measurements are included in the standard RTCP specification, in order to calculate those metrics, timestamp from sender and receiver is needed in the reports [39]. Sender Report (ST) timestamp is saved in the sender, meanwhile, the receiver returns the same timestamp in the Receiver Report (RR) that goes back to the origin. With those, we are able to calculate the RTT using the following Equation 4.

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

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 (4)

We can calculate the RTT using the data given by the *Stats API* in WebRTC, this metric is important to be monitored as it indicates the time delay on the link and the response of the media when received.

Calculating OWD requires both machines clock to be accurately synchronized and can be complicated, we try to accomplish this but usually OWD delay can be defined as $\frac{RTT}{2}$.

4.2.3 Throughput

Throughput is a key metric for testing the performance of WebRTC environments, this value is going to describe how much capacity of the link is taken by each *PeerConnection*. It is complex though, as there are still no fully functional QoS mechanisms implemented in WebRTC at this time. The throughput metric is going to provide bandwidth usage for video/audio in each direction, we can then use this value to get some quality metric in order to monitor the overall performance of the call. A sudden drop of the throughput usually mean that the bandwidth available for that *PeerConnection* has been drastically reduced, this produces artifacts and delays on the stream, or in the worst case, loss of communication between peers. In this

scenario ICE tries to renegotiate new candidates in order to obtain an alternative link for the connection and reestablish the streaming with the best possible throughput.

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 the result of discarding all the lost packets along the path, only packets that have been received are counted, goodput is a good metric to measure performance.

Typically, those metrics are calculated by extracting the information from the RTCP packets, in our case, we also rely on the Stats API included in WebRTC specification to obtain the amount of bytes and measurements to manually calculate rate by using JavaScript. Taking into account the last amount of bytes received, the actual amount and the time elapsed we are able to calculate an accurate value for the goodput.

$$\frac{\triangle Bytes_{received}}{\triangle Timestamp_{host}} \tag{5}$$

4.2.4 Inter-Arrival Time (IAT)

Due to latency on the path, packets can arrive at different times, congestion causes the increase and decrease in Inter-Arrival Time (IAT) between packets.

Congestion mechanisms in WebRTC use an adaptive filter that continuously updates the rate control of the sender side by using an estimation of network parameters based on the timing of the received frame. The actual mechanisms implement rate control by using IAT but other delay effects such as jitter are not captured by this model [33].

In WebRTC, IAT is given by the Equation 6, 7 and 8 [33].

$$\begin{aligned} &d(i) = t(i) - t(i-1) - (T(i) - T(i-1)) \\ &t\text{: arrival time} \\ &T\text{: timestamp time} \end{aligned} \tag{6}$$

Since the time t_s to send a frame of size L over a path with a capacity of C is approximately:

$$\begin{aligned} &t_s = \frac{L}{C} \\ &L\text{: Frame size} \\ &C\text{: Capacity of the link} \end{aligned} \tag{7}$$

The final model for the IAT in WebRTC can be simplified as:

$$d(i) = \frac{L(i) - L(i-1)}{C} + w(i) \tag{8}$$

Here, $w(i)$ is a sample from a stochastic process W which is given in function of C, the current cross traffic $X(i)$ and the send bit rate $R(i)$. If the channel congestion is high $w(i)$ will increase, otherwise $w(i)$ it is going to decrease decrease. Alternatively we can consider $w(i)$ equal to zero.

IAT is usually calculated in the receiver and reported to the sender to engage have sender-driven rate control.

4.3 Host metrics

Host metrics are measurements made locally by the host that affect the behavior of real time media communication. Those metrics do not need to be directly related to the network and can give information about the host performance when using WebRTC. They also provide information about timing and encoding for media in WebRTC sessions.

4.3.1 Resources

In WebRTC, we measure the local resource usage in the peers participating on a call, this metric is going to be very important for multiple peer topologies and resource demanding encoding.

CPU/RAM usage is observed in order to calculate the stress on the host system in WebRTC scenario. This information is crucial for the success of WebRTC in some environments that are demanding large amount of resources.

4.3.2 Setup time

We also measure the setup time required for every topology, with this we can possibly determine an average setup time for WebRTC connections. The setup time is calculated using local timestamps when building the *PeerConnection*Êobject.

Once the media form the remote stream arrives, we can subtract both timestamps, start of the *PeerConnection*Êand stream arrival, to check the elapsed time, this is done with STUN and TURN to check the difference in each NAT scenario.

4.3.3 Call failure

Call failure is an important metric that might help to decide wherever to ship a production application based on WebRTC or not. This also fully depend on NAT situations and we are going to calculate the average failed calls with STUN and TURN NAT techniques. This metric can define the success ratio for establishing WebRTC calls.

4.3.4 Encoding and decoding

WebRTC statistical API also provides real time information about the encoding and decoding bit rate of the video codec. This metric usually varies depending on the congestion mechanisms that change in different path conditions, it is a good

indicator for congestion and can give some hints about the way the encoding is done in WebRTC.

4.4 Summary of metrics

Performance metrics for WebRTC help us to determine the behavior of the link just by using information provided by the monitoring API of WebRTC, the goal of those metrics is to design better mechanisms to properly adapt the rate and response to the condition of the link.

Rate adaptation is a key metric used to deliver a good response in WebRTC calls and to closely study how rate perform in different topologies, some other metrics such as RTT, OWD and setup call directly affect the user experience on the call and should also be taken in consideration.

From the monitoring perspective we care about characterizing the performance and congestion control algorithm of WebRTC. This may also impact on the congestion control mechanisms and capacity sharing of the link.

5 Evaluation Environment

In this chapter we will describe the testbed we use for running all the WebRTC tests, the environment is described in Figure 13.

Some of the nodes carry specific software that enable some extra features or conditions that are required for deep testing.

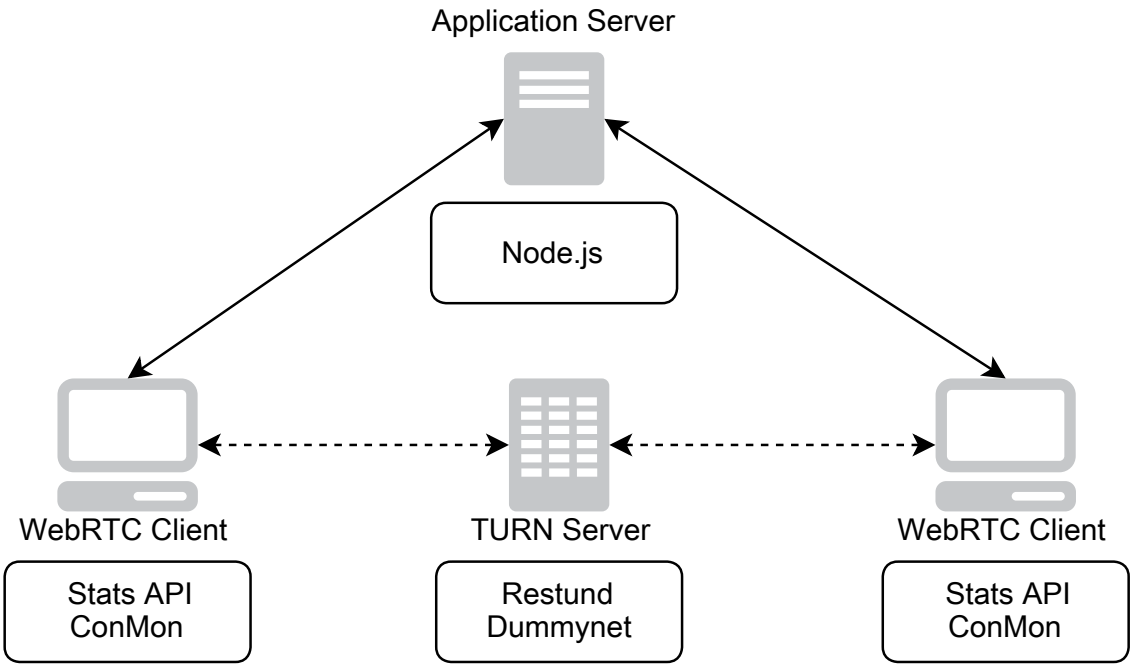


Figure 13: Description of simple testing environment topology for WebRTC.

With the testbed described in Figure 13 we are able to control the bottleneck link capacity, end-to-end latency, link loss rate and the queue size of the intermediate routers.

5.1 WebRTC client

WebRTC clients are virtual machines that run a lightweight version of Ubuntu (Lubuntu[1]) with 2GB of RAM and one CPU. This light version removes graphic acceleration providing better results in performance than compared with other distributions due to the virtualization of the graphic card.

Clients run Chrome Dev version 27.01453.12 as a WebRTC capable browser. To avoid unexpected results due to a bug in the *Pulse Audio* module of Ubuntu that controls audio input in WebRTC[2], calls are done with video only, the amount of audio transferred due to the echo cancelation systems can be neglected.

This client will load the web that runs on the Application Server to handle the WebRTC calls.

[1] https://wiki.ubuntu.com/Lubuntu

[2] https://bugs.launchpad.net/ubuntu/+source/pulseaudio/+bug/1170313

5.1.1 Connection Monitor

Connection Monitor [*ConMon*] is a command line utility that relies on the transport layer and uses *libpcap*[3] to sniff all the packets that to go a certain interface and port [40]. This utility is designed specifically to detect and capture RTP/UDP packets. *ConMon* detects and saves the header but discards the payload of the packet keeping the information we need for calculating our performance indicators.

Typically we run the *PeerConnection* between two devices and start capturing those packets using *ConMon* at each endpoint. The *PeerConnection* carries real media so the environment for testing is going to be a precise approach to a real scenario of WebRTC usage.

ConMon captures are be saved into different files allowing us to plot separate different stream rates and calculate other parameters such as delay with post processing. *ConMon* allows us to compare network layer and JavaScript API monitoring tools, as *ConMon* is working directly over the network interface and avoids all the processing that the browser internals do to send the stats to the JavaScript statistical API. Figure 14 represents one video stream from the same call as Figure 15 captured from the *ConMon* application.

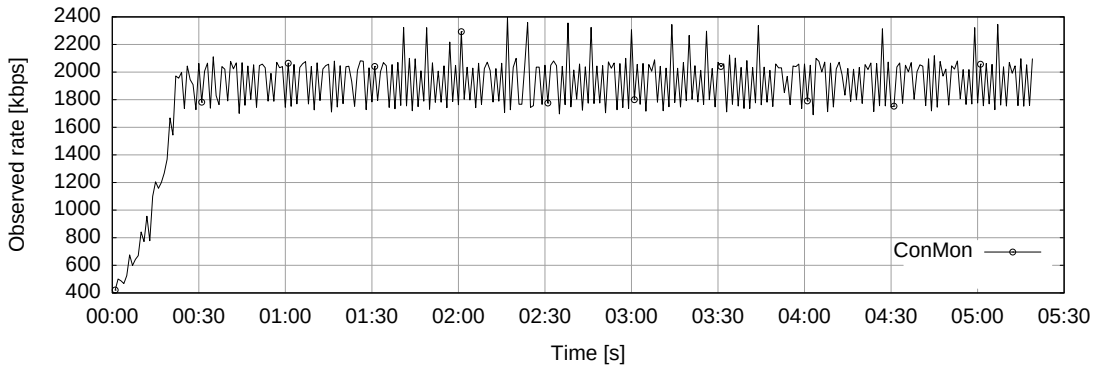


Figure 14: Point-to-point WebRTC video stream throughput graph using ConMon at the network layer.

The capture from *ConMon* is very accurate as it analyzes all the packets that go through an interface, this data is processed and averaged for a each period of one second.

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

ConMon provides us with higher accuracy by subtracting the timestamps from both endpoints of the same stream, to obtain a good result in this test we have to reduce the internal clock drift of both peers using systems such as Network Time Protocol daemon (NTPd) .

[3]http://www.tcpdump.org/

5.1.2 Stats API

WebRTC statistical API provides methods to help developers access the lower layer network information at the receiver, those methods return all different types of statistics and performance indicators that we use to build high level JavaScript *Stats API*. When using those statistics we process all the output data to obtain the metrics for WebRTC.

This system works in parallel with *ConMon*, both of them can provide similar results of some metrics and the comparison might be interesting to check the differences between the browser API and an interface layer capture. By communing both methods we can verify the results and accuracy of the metrics.

The method used for *Stats API* is the *RTCStatsCallback* that returns a JSON object that has to be parsed and manipulated to get the correct indicators, this object returns as many arrays as streams available in a *PeerConnection*, two audio and video [19] objects per *PeerConnection* when having a point-to-point call. This data is provided by the lower layers of the network channel extracting the information from the RTCP packets that come multiplexed in the same network port [38].

RTCStatsCallback is the mechanism of WebRTC that 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 in the following versions of the WebRTC API, methods involved in the *RTCStatsCallback* are available on the W3C editors draft [19].

We have built a high level *Stats API* that use those statistics from the *RTCStatsCallback* to calculate the RTT, throughput, loss rate and encoding/decoding rate for the different streams that are being processed. Those stats are saved into a file or sent as a JSON object to a centralized monitoring system. Our JavaScript API grabs any *PeerConnection* passed through the variable and starts looping a periodical iteration to collect those stats and, either plot them or save them into an array for post-processing. Figure 15 represents an example of a captured call between two browsers in two different machines, Mac and Ubuntu, the call was made over Wifi network with no firewall but with unknown cross traffic on it. The measures were directly obtained from the *Stats API* we built and post-processed using *gnuplot*[4].

Figure 15 plots the overall bandwidth of the call, this means that the input/output video and audio are measured together to check how much total bandwidth is being consumed over the duration of the call, as it is using RTCP packets to deliver the metrics to the *Stats API*, it takes a while to reach the average rate value until congestion mechanisms adapt the used rate to the network conditions. We can then plot all the different streams together to get an idea of how much bandwidth the *PeerConnection* is consuming.

5.1.3 Analysis of tools

Stats API and *ConMon* measure the same metrics but from different layers of the

[4]http://www.gnuplot.info/

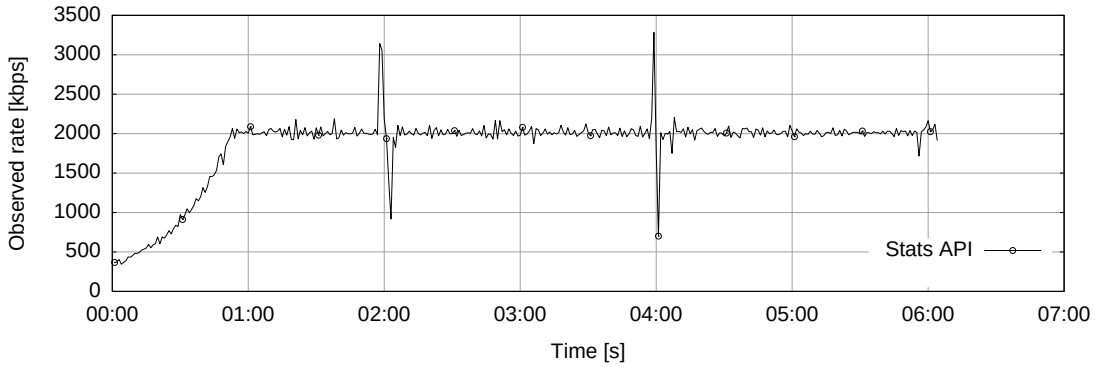


Figure 15: Point-to-point WebRTC video call total throughput graph using *Stats API* over public WiFi.

operating system, this provides us some extra information in order to see how the our high level *Stats API* works and if it is reliable and accurate.

However, due to the periodical capture method, the output can produce strange plots as the information regarding to the next data period could be stored in the previous one when processing the averaged data on the system. This is an accuracy problem that cannot be easily solved, when looking at the graph, it is important to observe if two peaks (positive and negative) get compensated by each other, this would mean that the data has not been allocated to the correct period when plotted. This accuracy error is a problem that can be observed when comparing both *ConMon* and *Stats API* capture in Figure **??**.

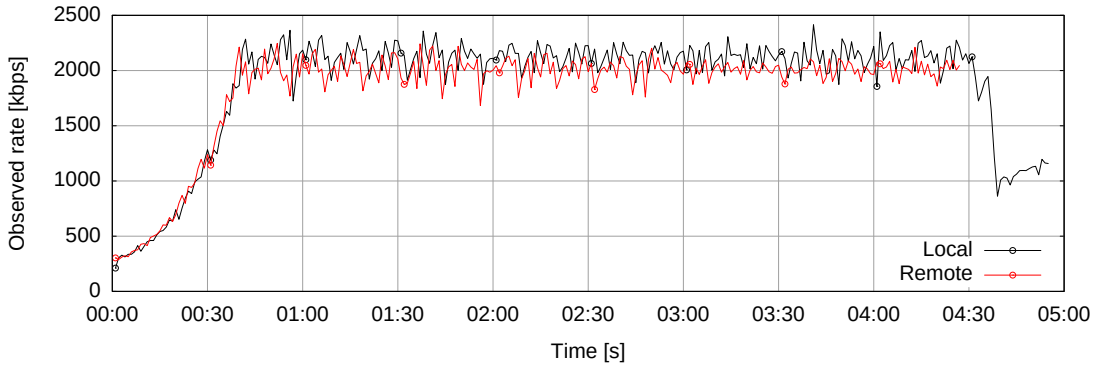
A second problem that we could face is the time it takes to the OS to process the stats form the RTCP packet and send them to the upper browser layer, at the receiver some of the stats are based on the current measurement of the metrics not in the RTP Receiver Report. Figure 16a and 16b plot two video streams being captured from Stats API and *ConMon*.

Figure 16 represents the incoming media stream from the other peer, we can see the little overhead that is not captured by the *Stats API* interface, as it just reads the bytes inside the payload of the packet. All the overhead is not considered when calculating the rate though *Stats API*. We can conclude that the real rate that WebRTC use is going to be defined by the result of *ConMon* instead of *Stats API*, but *Stats API* is going to be an accurate approach.

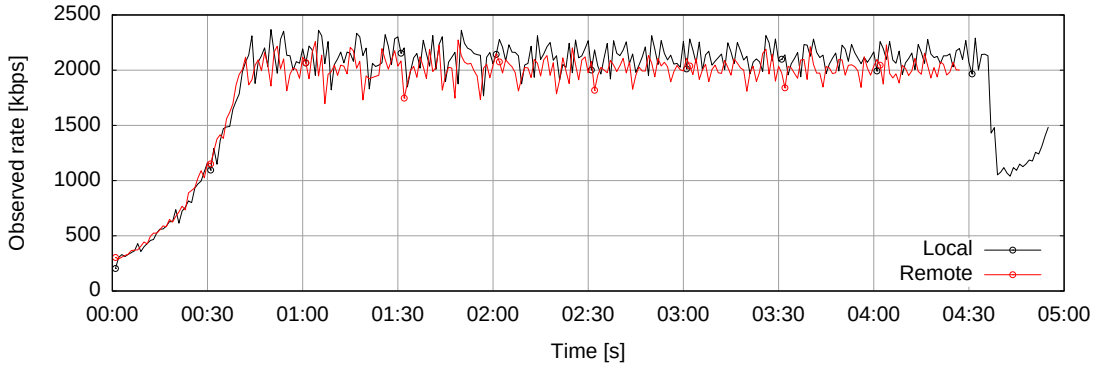
5.1.4 Automated testing

For our test 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 between all the iterations resulting in an accurate result.

In some environments, we won't be able to perform automated testing, when this happens the results won't be as accurate but they can provide a good approximation



(a) Incomming stream.



(b) Outgoing stream.

Figure 16: P2P video stream comparison between *ConMon* and *Stats API*.

to the averaged value.

One of the main issues when building a test scenario is the media provided to the *GetUserMedia* input, this media must be as close to reality as possible without using a real webcam. Google Chrome provides a fake video flag that can be activated by adding *–use-fake-device-for-media-stream*[5] parameter, this video though, does not produce enough rate for our purposes.

Figure 17 represents the approximate bandwidth that a real video call uses when sending media to another peer, that capture shows the same stream captured from the origin and receiver *StatsAPI* perspective. The adequate rate rises to 2000 Kbps. On the other hand, Figure 18 represents the scenario but using the built-in fake video in both clients, the rate for this case drops to an average of 250 Kbps.

Both figures (17 and 18) print one unique stream, identified with the Synchronization Source identifier (SSRC) , but from the sender and receiver perspective, *LV* identifies the source capture and *RV* the receiver stream rate.

Comparing global output from Figures 17 and 18, we can see that the obtained rate is very different concluding that we cannot use *–use-fake-device-for-media-stream* flag for our testing environment. The reason is that Google Chrome uses a bitmap system to draw the figures and components that will be rendered

[5]http://peter.sh/experiments/chromium-command-line-switches/

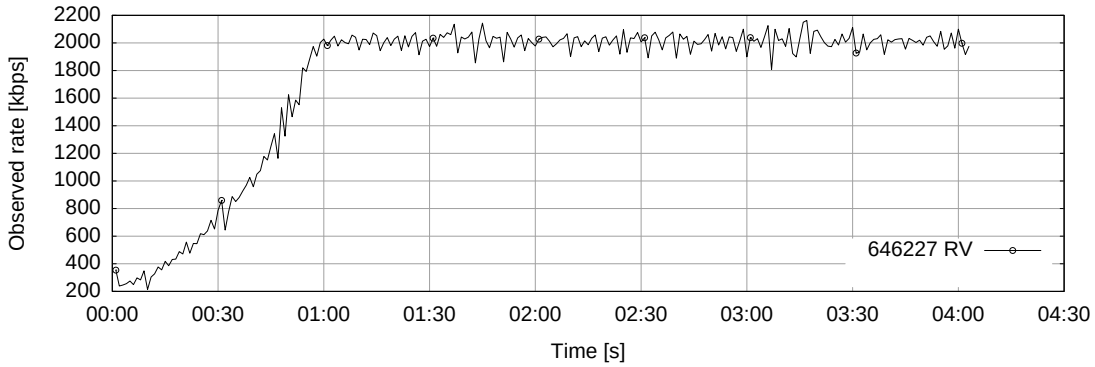


Figure 17: Video stream rate with SSRC 0x646227 captured using *Stats API* and webcam input.

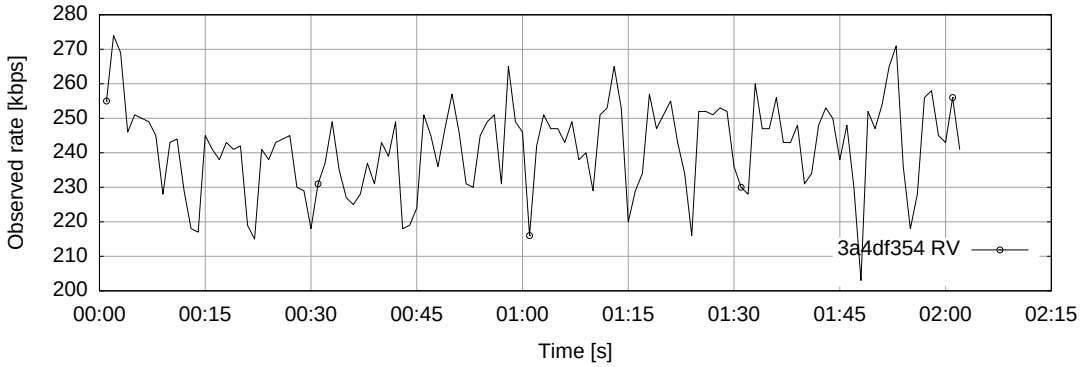


Figure 18: Video stream rate with SSRC 0x3a4df354 captured using *Stats API* and Chrome default fake video input.

in the video tag and sent over the *PeerConnection*, this means that the amount of encoding and bandwidth used will be low compared to a real webcam as the media sent over with fake video is minimum.

To address this issue of the media streaming for our automated devices, we have built a fake input device on the peers, the procedure is described in Appendix A.

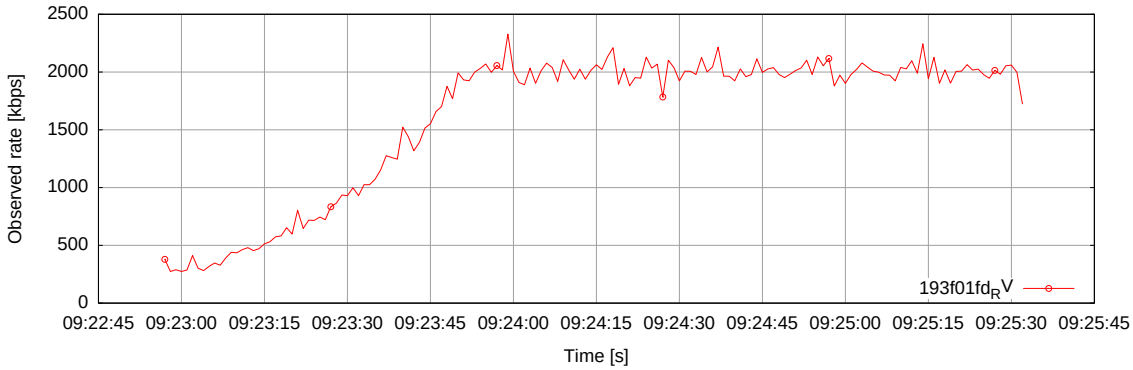


Figure 19: Video stream bandwidth using V4L2Loopback fake YUV file.

Figure 19 represents the bandwidth of a fake video stream measured by our *Stats API* using an YUV[6] video captured from a Logitech HD Pro C910 as source, resolution is 640x480 at a frame-rate of 30 fps.

Results can be compared between Figure 19 and 17, both average rate output is approximately 2000 Kbps, which means that this procedure is a good approach to a real webcam.

The combination of the previous fake video setup and multiple Secure Shell scripts enables the automation mechanisms to run multiple tests without the need of multiple physical devices.

5.2 TURN Server

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

The TURN daemon we use is called *Restund*, which has been proven to be reliable for our needs, this open source STUN/TURN server works with *MySQL* database authentication [41]. 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. Content of method *restund_get_ha1* has to be replaced with the following line of code, where XXX is username and YYY the password we use for the TURN configuration.

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 candidates we need to replace the WebRTC API server identification with our TURN machine 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 previous object is provided to the *PeerConnection* object enabling the use of TURN.

The IP address points to our TURN server and 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 may override TURN, to force the usage of TURN candidates we need to drop all candidates that do not force the use of the relay.

[6]YUV is a color space that encodes video taking human perception into account, typically enabling transmission errors or compression artifacts to be more efficiently masked by the human perception than using RGB-representation.

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* is 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 own method *sendMessage* through *WebSockets* or similar polling methods. In this code, we are dropping all candidates except the ones containing the option *relay* on it, those are the candidates that force the *PeerConnection* 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 WebRTC peers.

5.2.1 Dummynet

To evaluate the performance of WebRTC we may modify the conditions of the network path to imitate some specific environments. 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 [42].

Dummynet is an standard tool for some Linux distributions and OSX [42]. In order to get appropriate results we need to apply the *Dummynet* rules in the TURN server, this machine will forward all the WebRTC traffic from one peer to the other being transparent for both ends.

The real goal of using TURN in WebRTC is to bypass some restrictive Firewalls that could block the connection, in our case, this works as a way to centralize the traffic flow through one unique path that we can monitor and modify. From the performance perspective, when not adding any rules to the TURN, the traffic and response of WebRTC is normal without the user noticing any difference.

Some problems arise when using *Dummynet* in our scenario, we will use *VirtualBox*[7] machines for some testing and for running TURN instance, read Appendix B for the fixes in *Dummynet* configuration for virtual machines.

[7] VirtualBox is an x86 virtualization software package.

5.3 Application Server

Our application server runs the Node.js [8] instance to handle the WebRTC signaling part, this machine uses Ubuntu with a domain name specified as *dialogue.io*.

This app is a common group working application that allow people to chat and video call at the same time in their own private chat rooms, we have modified it to build an specific room for our tests, this instance simply allow two users that access the page to automatically call each other and start running the JavaScript code with the built-in *Stats API*.

Most of this application is coded using JavaScript APIs and uses WebSocket [9] protocol to handle the signaling messages from peer to peer.

The web browsers in our clients stablish an HTTPS connection between the Application Server to download the web page that contain the *call.js* file that executes the WebRTC features.

5.4 Summary of tools

Using all the previous mentioned tools together we are 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 use some special scripts that measure and extract the information we require form the captures, some of them are explained in Appendix C.

[8] http://nodejs.org/
[9] http://socket.io/

6 Performance evaluation of WebRTC

In this chapter we study how WebRTC performs in different network environments using topologies previously described in chapter 3.

All tests are deployed using the testbed described in chapter 5 and Figure 13. We have each component running in individual virtual machines, such as the clients, the TURN server and the application server that establishes the negotiation between calls.

To introduce different impairments in the bottleneck we force some of the sessions to run through the TURN server and we use *Dummynet* to emulate the variation in link capacity, latency, intermediate router queue length. We also use the *Gilbert-Elliott Model* to model packet loss [43] [44].

WebRTC-enabled browsers implement a congestion control algorithm proposed by *Google* called *Receiver-side Real-time Congestion Control (RRTCC)* [33].

In order to properly adapt the rate to the link conditions, RRTCC need to use two components: sender-side and receiver-side.

This protocol enables congestion control from the receiver side, in the following sub-sections we will discuss which important features RRTCC enable. The receiver estimates the usage of the link based on inter-arrival time (IAT) of the incoming packets. Three different states can be given in a link: overuse, stable and under-use. The current receiver estimate, $A_r(i)$ is calculated as follows:

$$A_r(i) = \begin{cases} 0.85 \times RR & \text{overuse}, m(i) > 0 \\ A_r(i-1) & \text{stable}, m(i) = 0 \\ 1.5 \times RR & \text{under-use}, m(i) < 0 \end{cases} \tag{9}$$

The receiving endpoint sends an RTCP feedback message to the sender containing the Receiver Estimated Media Bitrate (REMB) [45], A_r at 1s intervals.

Once the feedback message is received, the sender side uses the TCP Friendly Rate Control (TFRC) mechanism to estimate the sending rate based on the loss rate, RTT and bytes sent [46]. This equation specifies that the rate should not vary when the loss rate is stable between 2-10%, but must be modified otherwise. If no feedback message is received, the rate is halved on the sender side. The new sender available bandwidth (A_s) is calculated using Equation 10, being p the packet loss ratio.

$$A_s(i) = \begin{cases} A_s(i-1) \times (1 - 0.5p) & p > 0.10 \\ A_s(i-1) & 0.02 \leq p \leq 0.1 \\ 1.05 \times (A_s(i-1) + 1000) & p < 0.02 \end{cases} \tag{10}$$

Furthermore, the $A_s(i)$ calculated should be between the rate estimated by the TFRC equation and the pervious REMB value received from the receiver report. This algorithm does not react to losses under 10% and increases the rate by 5% when having losses between 2-10%. Alongside with this mechanism, WebRTC also includes Forward Error Correction (FEC) mechanism, this technique generates FEC packets to protect the media stream when losses appear on the path.

Other congestion control mechanisms have been proposed with the goal to match the rate with available capacity while maintaining a good user experience. Instead of just relaying in RTT and loss, Garudadri *et al.* [47] also use the receiver play out buffer to detect the utilization of the link. Singh *et al.* [48] uses frame inter-arrival time and play out buffer size to adjust the rate to the link capacity. Zhu *et al.* [49] use ECN and loss rate to get a more accurate metric on the loss during the path and adjust the rate control based on the result. O'Hanlon *et al.* [50] proposes a congestion mechanism for cross-traffic environments, using a delay-based estimation it should adapt the rate when having similar traffic on the network. In case off TCP cross-traffic, it uses a windowed-approach to adapt the rate. They switch the operational modes by using a threshold on the observed end-to-end delay.

While those congestion control mechanisms have been proposed, they have not been implemented yet in any WebRTC browser.

6.1 Experimental Methodology

In the following sub-sections we evaluate different scenarios for WebRTC:

- Single RTP WebRTC session with different networks constraints
- Single RTP WebRTC flow with TCP cross traffic
- Multiple RTP flows
- WebRTC group calls using full mesh
- Wireless scenario for WebRTC
- Interoperability tests between different browser providers
- Mobile environment performance

Each scenario is evaluated by running multiple benchmarks of tests, most of the tests run up to 15 times to derive statistical significance, lasting 300 seconds each iteration.

6.2 Effects of Varying Bottleneck Link Characteristics

We have performed different tests in a bidirectional RTP scenario to study how the WebRTC applications handles different bottleneck conditions. We vary network characteristics and observe the RRTCC performance.

6.2.1 Non-constrained link

Firstly, we are going to proceed with a sample test in a non-constrained link to check the performance of WebRTC.

Figure 20 represents the average rate of every iteration of the test in a wired network without any link constraint, the average rate obtained in the test is 1949.7±233

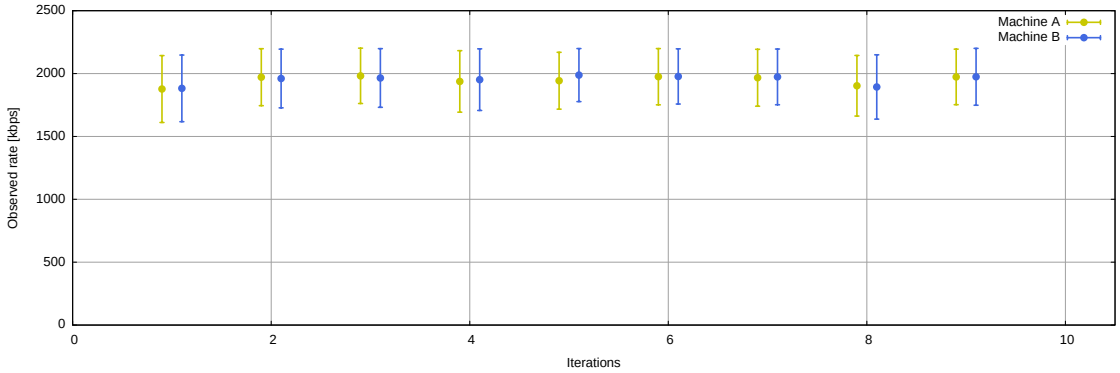


Figure 20: Rate average and deviation for non-constrained link in each iteration.

Kbit/s, we can conclude that a standard rate for a video call in a non-constrained link using WebRTC is approximately 2 Mbit/s. Furthermore, obtained delay is 5.1±1.5 ms and RTT is approximately 9.5 ms, those results can be taken as standard for a non-conditioned WebRTC call.

A summary of results is available in Table 2, besides the network performance we are also tracking the call failure rate, considering all those calls go through the bottleneck TURN server we might be able to approximate the success rate when establishing calls in WebRTC.

Setup time is evaluated with the time it takes since the creation of the *PeerConnection* object until the media stream from the other peer arrives, this value defines the time it takes for the user to start the communication, in an optimal environment it takes approximately 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
Rate (Kbit/s)	1947.61±232.75	1951.76±234.5	1949.7±233.62
RTT (ms)	9.49±2.11	9.64±2.71	9.57±2.41
OWD (ms)	4.84±1.5	5.4±1.53	5.12±1.52
Residual Loss (%)	0.012	0.01	0.011
Packet Loss (%)	0.012	0.01	0.011
Setup time (ms)	1436.33±25	1447.44±22.71	1441.88±24.04

Table 2: P2P metrics output for WebRTC call with no link restriction.

Delay values in Table 2 are averaged using all the *ConMon* data obtained for each stream in all iterations, thus it is an approximate delay value it might not be representative of the exact delay occurred during the call. Considering the example in Figure **??** we can see that the delay can is variable, this means that the averaged OWD may not be the most appropriate metric to measure the time delay user experience. In order to evaluate the behavior of WebRTC in delay, we have two different approaches, the mean delay with deviation and delay distribution of all calls.

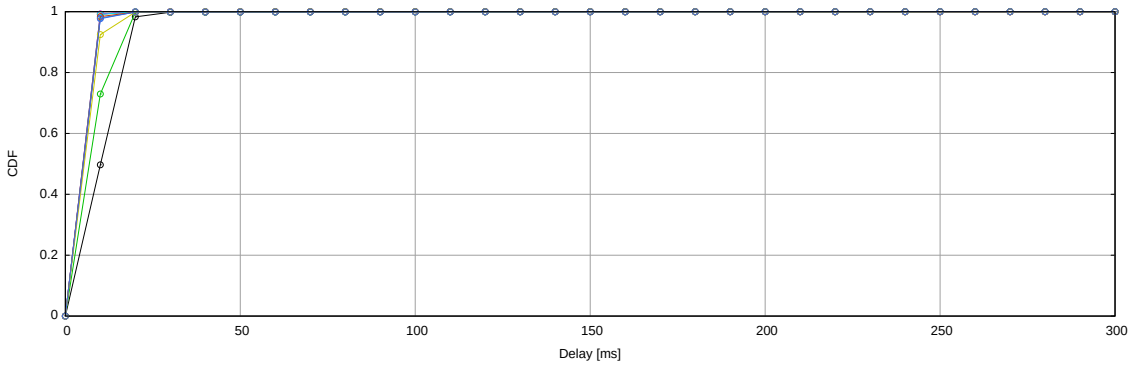


Figure 21: Delay distribution for each P2P iteration with no link constraints.

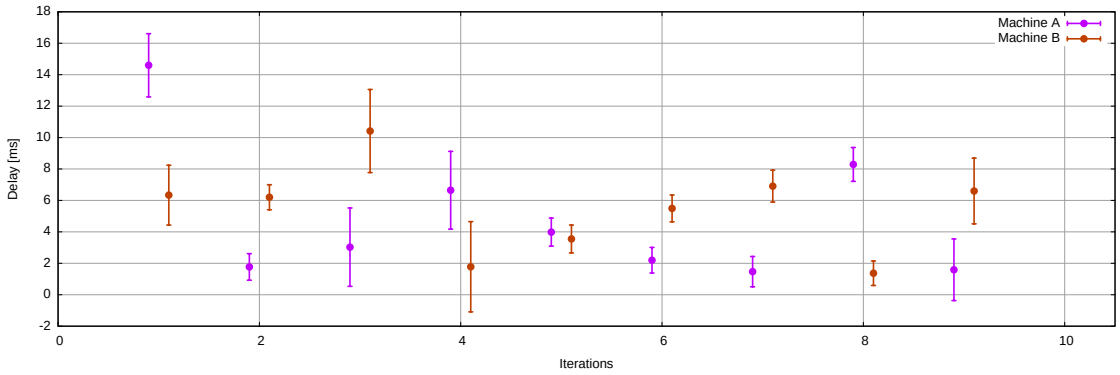


Figure 22: Mean and deviation for OWD in each P2P iteration with no link constraints.

Figure 22 represents the mean and deviation for the delay calculated in each iteration, this delay is calculated by using the arrival timestamp for each packet with the capture done in both peers with *ConMon*. We run a NTPD daemon to adjust the drift on the clock and sync both machines. Figure 22 represents the delay output in a clearer way than the averaged result between all iterations, the difference in each iteration is small resulting in about 10ms difference between the best and worst case.

In Figure 21, the distribution is given by the amount of packets whose delay is in a certain range of time, they are counted by batches of 10ms with an adaptable maximum range for each scenario. Most of the packets run with less than 25ms delay in all the iterations for the non-constrained test. The user experience with this small amount of delay that do not vary on time is barely negligible. Figures 22 and 21 try to evaluate the delay response for a specific scenario, this can be difficult as the real time response changes in all iterations depending on the actual of the link, we use the delay distribution and OWD mean to evaluate the delay response.

However, with the captures performed in *ConMon* and *Stats API* we can also evaluate one specific stream delay response if we need more information regarding a unique iteration. Figure 23 represents the OWD response along the call of one specific video stream, is easy to see that there is some small variations that may

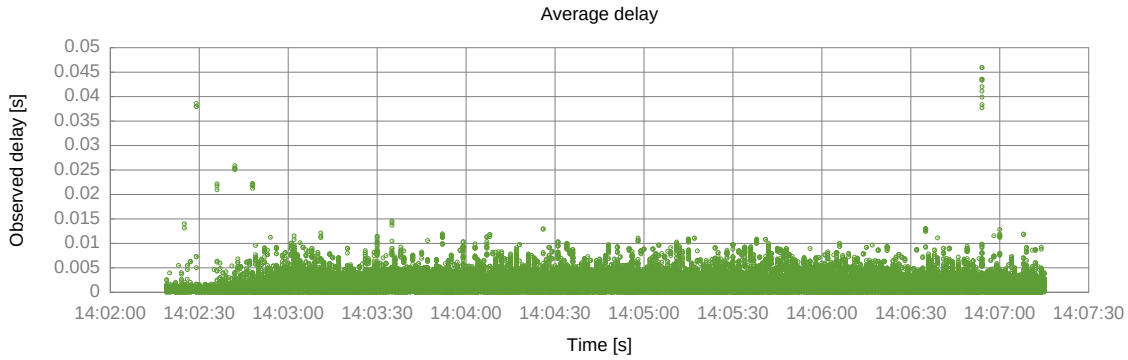


Figure 23: OWD response during the call for one video stream for a non-constrained link call.

affect the call but the overall delay value is stable.

6.2.2 Variable latency

We have benchmarked tests with different one-way latencies, 50, 100, 200 and 500ms. We should be able to observe that the increase of latency reduces the average media rate during the call.

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.

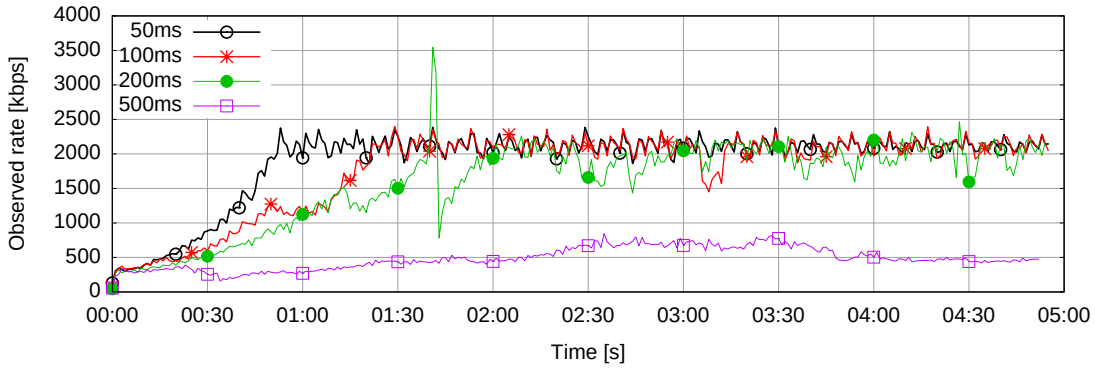


Figure 24: Media rate variation in receiver due to different bottleneck latency.

Table 3 represents the average bandwidth response to the different latency conditions, one important fact is that the loss observed in all situations is barely negligible. The RTT is also not noticing any unexpected change. On the other side, we can observe in Figure 24 and Table 3 that the increase in latency drastically reduces the media rate.

The problem with WebRTC relies in the usage of RTP over UDP for packet transport as UDP does not carry the sophisticated congestion control mechanisms

	Metrics	Machine A	Machine B	Overall
50 ms	**Rate (Kbit/s)**	1909.31±258.09	1917.81±251.62	1913.56±254.86
	OWD (ms)	102.35±1.29	102.67±1.58	102.51±1.44
	Residual Loss (%)	0.02	0.09	0.05
	Packet Loss (%)	0.02	0.09	0.05
100 ms	**Rate (Kbit/s)**	1516.07±263.43	1453.94±272.79	1485±268.11
	OWD (ms)	202.82±2.94	202.32±3.05	202.57±3
	Residual Loss (%)	0.1	0.02	0.06
	Loss (%)	0.1	0.02	0.06
200 ms	**Rate (Kbit/s)**	503.71±116.45	617.92±142.69	560.82±129.57
	OWD (ms)	402.06±3.3	401.75±3.31	401.91±3.33
	Residual Loss (%)	0.3	0.35	0.33
	Packet Loss (%)	0.36	0.43	0.4
500 ms	**Rate (Kbit/s)**	303.58±59.22	207.77±32.48	255.67±45.85
	OWD (ms)	1001.3±3.8	1001.41±4.09	1001.36±3.99
	Residual Loss (%)	0.6	0.09	0.35
	Packet Loss (%)	0.63	0.1	0.37

Table 3: Summary of averaged results with different latency conditions.

that TCP does. When handling real time media, modifying the encoding rate to accommodate the varying bandwidth can be difficult and slow.

Low latency networks will play a big role when using WebRTC in mobile devices where the ability to react to latency and packet losses will be crucial for its success against other alternatives.

6.2.3 Lossy environments

This chapter evaluates the response of WebRTC calls in lossy environments, those situations can be produced in mobile environments with low coverage or when having packet drops in any link due to heavy congestion. Discarding packets in the peers for large delay also produces losses that can affect the call.

Losses in WebRTC directly affect the quality of the media that is sent over the path, when having heavy loss, artifacts appear on the video and the user experience degrades exponentially.

We have tested a bi-directional bottleneck call with 1, 5, 10 and 20% of packet loss, according to the results in Table 4, we can see that an increase of loss rate decreases the media rate. The RRTCC Google algorithm [33] does not react for losses under 10%. Figure 25 represents the rate variation for the different packet losses, being clear the action of the RRTCC in different cases.

In Table 4, there are two different packet loss rate, the *Packet Loss* indicates the packet loss along the link, meanwhile *Residual Loss* refers to the losses after the error correction performed by RTP FEC mechanism. When having losses in a WebRTC RTP stream, the system may ask for a retransmission of the lost packet

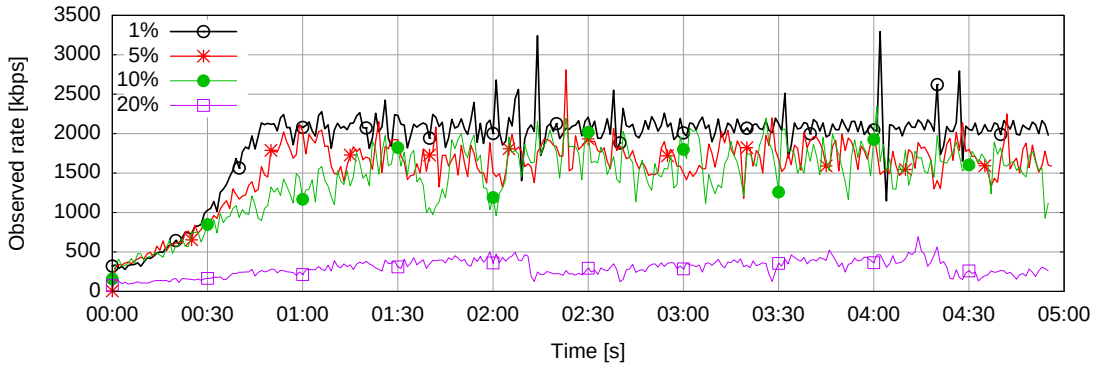


Figure 25: Media rate variation in receiver due to different bottleneck packet loss.

in some occasions, if the delay is small it can be possible to forward them in real time, this metric is then referenced as *Residual Loss*. Those type of losses will be the ones affecting directly the data received on the peer so a low value will barely affect the quality of the call [51].

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.

	Metrics	Machine A	Machine B	Overall
1%	**Rate (Kbit/s)**	1913.59±252.11	1880.24±261.46	1986.91±256.78
	OWD (ms)	4.08±1.79	4.03±1.93	4.06±1.86
	Residual Loss (%)	0.13	0.05	0.09
	Packet Loss (%)	2.04	1.96	2
5%	**Rate (Kbit/s)**	1609.65±158.46	1527.74±178.52	1568.74±178.52
	OWD (ms)	3.72±1.82	3.26±1.76	3.49±1.79
	Residual Loss (%)	0.2	0.27	0.23
	Loss (%)	9.72	9.82	9.77
10%	**Rate (Kbit/s)**	1166.7±145.96	1114.94±177.88	1140.82±161.92
	OWD (ms)	3.17±3.8	3.12±2.67	3.14±3.24
	Residual Loss (%)	0.58	0.41	0.49
	Packet Loss (%)	18.98	19.05	19.02
20%	**Rate (Kbit/s)**	333.34±65.99	295.46±57.98	314.4±61.98
	OWD (ms)	2.65±4	2.78±4.06	2.71±4.03
	Residual Loss (%)	2.69	2.17	2.43
	Packet Loss (%)	36.08	35.95	36.01

Table 4: Rate, OWD and loss averaged results for different packet loss constraints on the link.

Observing Table 4, we can see that the packet loss in the transport layer, where *ConMon* is capturing the RTP packets, is higher than the configured in *Dummynet*. This happens due to the probabilistic model of *Dummynet* for packet drop, this

mechanism is totally random being not a good approach to reality in some cases and could led to unexpected behavior [52]. However in wired environments, packet drops are usually due to queue overflows, queue management schemes, or routing problems. Radio links add noise and interference as other potential causes of drops. Those circumstances are not easily reproducible in testing environments.

On the other side, the sender calculates the rate based on the receiver report that arrives from the other peer. 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.

6.2.4 Loss and delay

Based on the two previous sections we introduce both loss and latency in the bottleneck link. We have set 10% packet loss with different delays such as 25ms, 50ms, 100ms and 200ms. Table 4 shows 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 RRTCC to handle congestion in those environments.

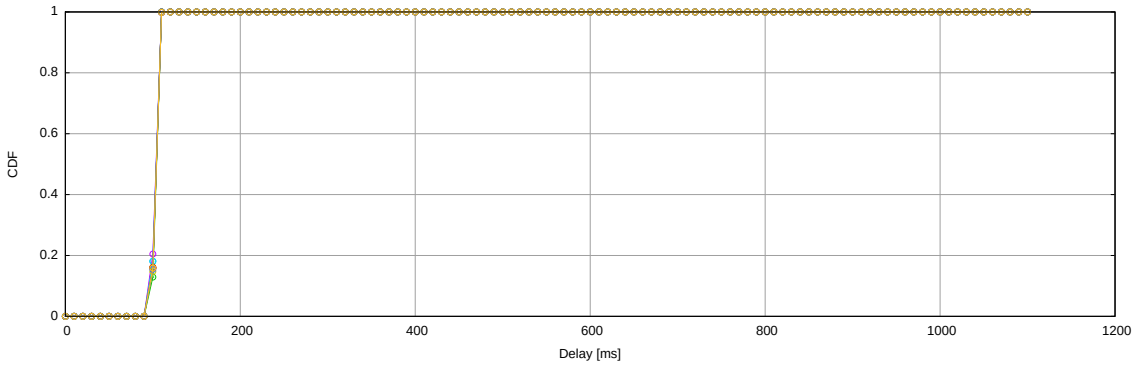
	Metrics	Machine A	Machine B	Overall
25 ms	**Rate (Kbit/s)**	72.59±18.54	70.96±18.09	71.78±18.32
	OWD (ms)	87.64±16.12	77.67±16.15	82.66±16.14
	Residual Loss (%)	6.5	7.02	6.77
	Packet Loss (%)	18,7	19.12	18.91
50 ms	**Rate (Kbit/s)**	50.32±15.39	60.36±18	57.19±16.63
	OWD (ms)	101.16±0.56	101.36±0.56	101.26±0.56
	Residual Loss (%)	19.29	11.32	15.49
	Loss (%)	26.46	19.13	22.97
100 ms	**Rate (Kbit/s)**	63.3±19.29	64.82±20.95	64.06±20.12
	OWD (ms)	201.59±3.03	201.36±6.09	201.48±4.56
	Residual Loss (%)	10.91	10.82	10.87
	Packet Loss (%)	18.81	18.67	18.77
200 ms	**Rate (Kbit/s)**	66.89±20.12	65.66±19.63	66.27±19.87
	OWD (ms)	403.73±25.67	403.93±31.71	403.83±28.69
	Residual Loss (%)	11.35	12.01	11.68
	Packet Loss (%)	18.59	19.07	18.83

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

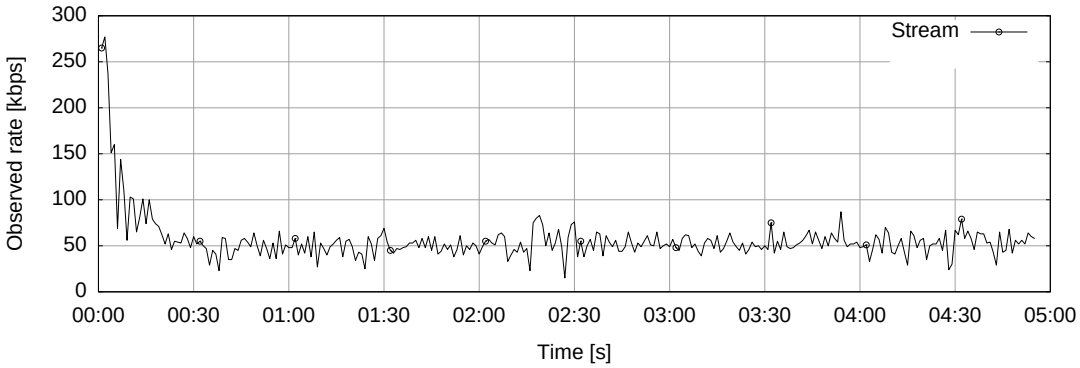
Compared with Table 4 and Table 3 , we can observe that RRTCC cannot handle the retransmissions of packets using FEC. The residual loss increases in all latencies from 0.49% to 6.77% in the best latency situation while the overall packet loss remains similar ≈19%. Figure 26b shows how the sending rate reduces when facing loss and delay on the same path being RRTCC unable to fix this issue with it's

internal mechanisms. On the other side, Figure 26a indicates that meanwhile the rate is reduced the RTT keeps constant during all the duration of the call at 100ms.

If we study the way WebRTC calculates the rate in this situation we can see that the sender decision will be based on the RTT, packet loss and available bandwidth that is estimated from the receiving side using Equation 10 [33]. 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 10, the sender rate is always limited by the TCP Friendly Rate Control (TFRC) formula that is calculated using delay an packet loss ratio together [46].



(a) CDF of delay distribution for 100ms latency.



(b) Remote stream bandwidth for 10% packet loss rate and 50ms delay.

Figure 26: Bandwidth and mean for 1 Mbit/s with multiple queue sizes

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

6.2.5 Bandwidth and queue variations

In this section we vary the queue size at an intermediate router in order to observe the impact of the performance in the RRTCC algorithm. For this test we have selected to run 500 Kbit/s, 1 and 5 Mbit/s throughput rate with different queue lengths from 100 ms, 500 ms, 1s and 10 s. In total we have run 12 different tests with ten iterations each.

We calculate the queue size in function of time, being this the amount of time the packet will remain in the router before being discarded. However, the result is given in number of packets using Equation 11. The MTU size in this test is set to 1500 bytes.

$$Queue(packets) = \frac{Bandwidth(Bits)}{8 \times MTU} \times Queue(seconds) \tag{11}$$

For example, when setting a throughput of 1Mbps and 1s queue depth, a router would be able to handle 83 packets as a total queue length. A queue length of 100ms represents a short queue meanwhile a 10s queue identifies a buffer-bloated queue.

Tables 6, 7 and 8 show the result for the rates and queue sizes and bandwidth used for the tests.

After running the tests we can state that the overall Average Bandwidth Utilization (ABU) for scenarios with 500 Kbit/s, 1 and 5 Mbit/s is ≈0.4 (40%). Furthermore we can assure that varying the queue lengths does not directly affect the performance of RRTCC.

500 Kbit/s				
Queue	Metrics	Machine A	Machine B	Overall
100 ms	**Rate (Kbit/s)**	243.17±35.59	154.85±30.41	199.01±33
	OWD (ms)	39.19±11.85	39.69±17.34	39.44±14.59
	Residual Loss (%)	1.94	1.87	1.91
	Packet Loss (%)	3.4	4.57	3.99
500 ms	**Rate (Kbit/s)**	203.71±26.81	175.1±20.84	189.41±23.83
	OWD (ms)	35.92±30.57	35.43±30.34	35.67±30.34
	Residual Loss (%)	0.34	0.24	0.29
	Loss (%)	0.78	0.74	0.76
1 s	**Rate (Kbit/s)**	214.81±16.69	211.59±15.74	213.2±16.22
	OWD (ms)	47.43±61.2	48.18±62.23	47.8±62.23
	Residual Loss (%)	0.33	0.43	0.38
	Packet Loss (%)	0.59	0.63	0.61
10 s	**Rate (Kbit/s)**	210.54±14.92	218.56±13.84	214.55±14.38
	OWD (ms)	86.42±148.97	89.34±150.5	87.88±149.73
	Residual Loss (%)	0.07	0.06	0.06
	Packet Loss (%)	0.06	0.05	0.06

Table 6: Averaged results with different queue configurations and 500 Kbit/s bandwidth constraint.

However, we can study the result of the test performed with 1 Mbit/s throughput bottleneck limitation, as the maximum standard bandwidth for WebRTC is approximately 2 Mbit/s, setting 1 Mbit/s throughput limitation forces RRTCC to adapt the outgoing rate to the link conditions.

Figure 29 represents the bandwidth and mean plotted for all the different latencies with 1 Mbit/s limitation. We can see that the average rate response varies in

1 Mbit/s				
Queue	Metrics	Machine A	Machine B	Overall
100 ms	**Rate (Kbit/s)**	436.72±72.7	289.02±63.66	362.87±68.18
	OWD (ms)	30.91±11.75	31.94±12.05	31.42±11.9
	Residual Loss (%)	1.22	0.37	0.79
	Packet Loss (%)	1.77	1.07	1.42
500 ms	**Rate (Kbit/s)**	323.59±82.33	285.21±73.71	304.4±78.02
	OWD (ms)	24.72±10.69	28.55±11.04	26.63±10.86
	Residual Loss (%)	0.16	0.08	0.12
	Loss (%)	0.16	0.08	0.12
1 s	**Rate (Kbit/s)**	401.41±58.14	347.86±59.15	374.64±58.64
	OWD (ms)	28.94±8.7	26.02±8.9	27.48±8.8
	Residual Loss (%)	0.06	0.04	0.05
	Packet Loss (%)	0.06	0.04	0.05
10 s	**Rate (Kbit/s)**	481.08±31.01	395.55±32.15	438.31±31.58
	OWD (ms)	26.83±7.08	27.5±7.56	27.16±7.32
	Residual Loss (%)	0.04	0.05	0.04
	Packet Loss (%)	0.04	0.05	0.04

Table 7: Averaged results with different queue configurations and 1 Mbit/s bandwidth constraint.

small amount of bandwidth but with large deviation in each iteration, when having 500ms and 1s queue size (29c) we have larger deviation in means of packets being buffered in the relay. Otherwise, when the queue size reduces to 100ms (29d) 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. Furthermore, for the 100ms test, the delay is not constant during the test with a maximum delay of 100ms.

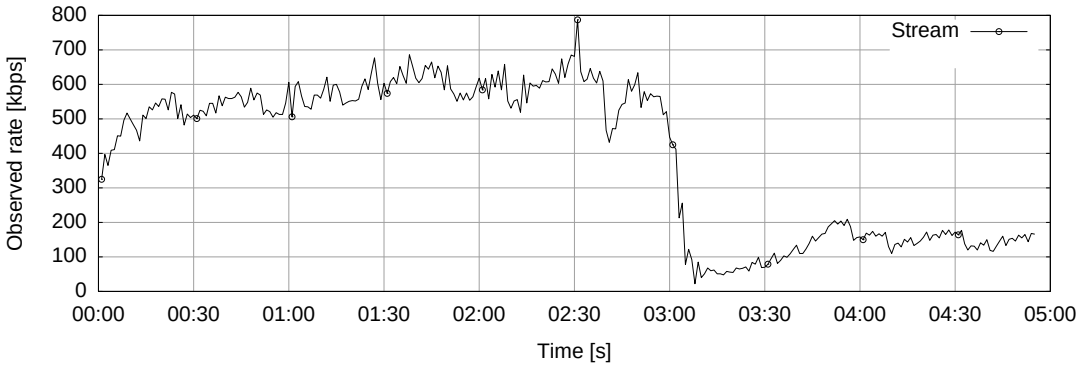


Figure 27: Remote stream bandwidth for 1 Mbit/s and 500ms queue size.

Delay experience with small queue sizes will be worst for the user experience,

5 Mbit/s				
Queue	Metrics	Machine A	Machine B	Overall
100 ms	**Rate (Kbit/s)**	1963.15±227.1	1979±211.46	1965.95±224.02
	OWD (ms)	20.38±4.13	19.89±4.19	20.13±4.16
	Residual Loss (%)	0.02	0.01	0.02
	Packet Loss (%)	0.02	0.01	0.02
500 ms	**Rate (Kbit/s)**	1595.41±266.41	1565.36±264.06	1580.39±265.24
	OWD (ms)	18.15±14.06	17.83±12.99	17.99±15.52
	Residual Loss (%)	0.03	0.02	0.03
	Loss (%)	0.03	0.02	0.03
1 s	**Rate (Kbit/s)**	1919.25±231.35	1922.09±237.41	1920.67±243.38
	OWD (ms)	19.64±5.38	17.25±5.76	18.45±5.57
	Residual Loss (%)	0.02	0.01	0.02
	Packet Loss (%)	0.02	0.01	0.02
10 s	**Rate (Kbit/s)**	1595.41±266.41	1565.36±264.06	1580.39±264.24
	OWD (ms)	18.15±14.06	17.83±12.99	17.99±13.52
	Residual Loss (%)	0.03	0.02	0.03
	Packet Loss (%)	0.03	0.02	0.03

Table 8: Averaged results with different queue configurations and 5 Mbit/s bandwidth constraint.

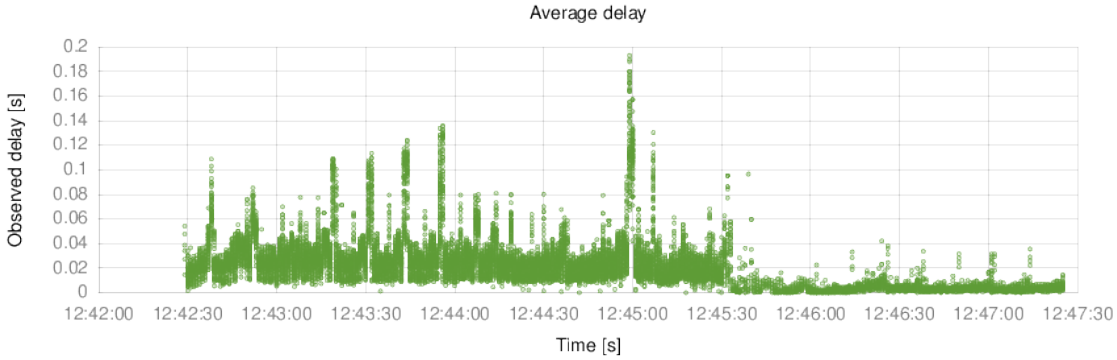


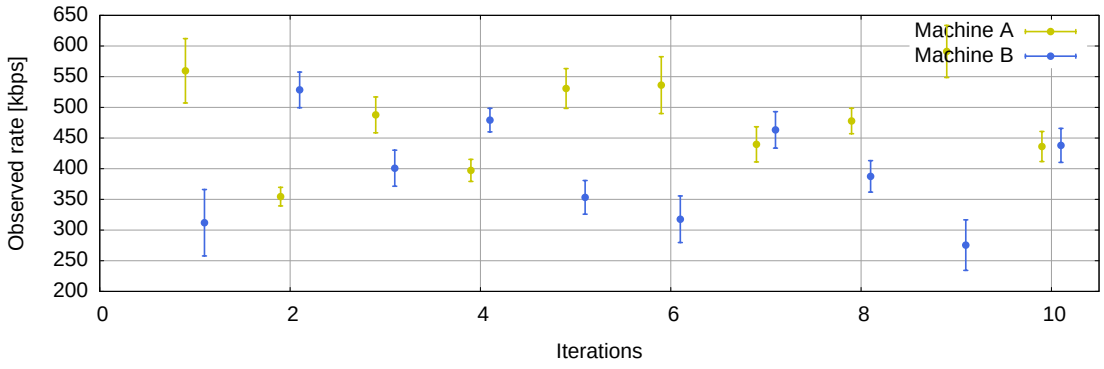
Figure 28: Stream delay for 1 Mbit/s and 500ms queue size.

we might experience unexpected delays that RRTCC won't be able to handle, when having larger queue sizes we won'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 RRTCC is able to adapt to low rate 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.

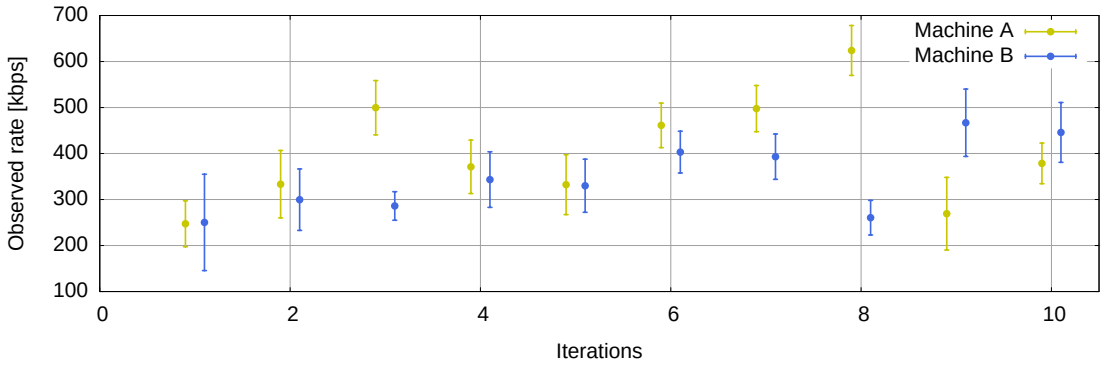
RRTCC stabilizes the rate until the amount of delay triggers the algorithm to fit the new queue state. Figure 27 and 28 show the rate and delay for the same

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.

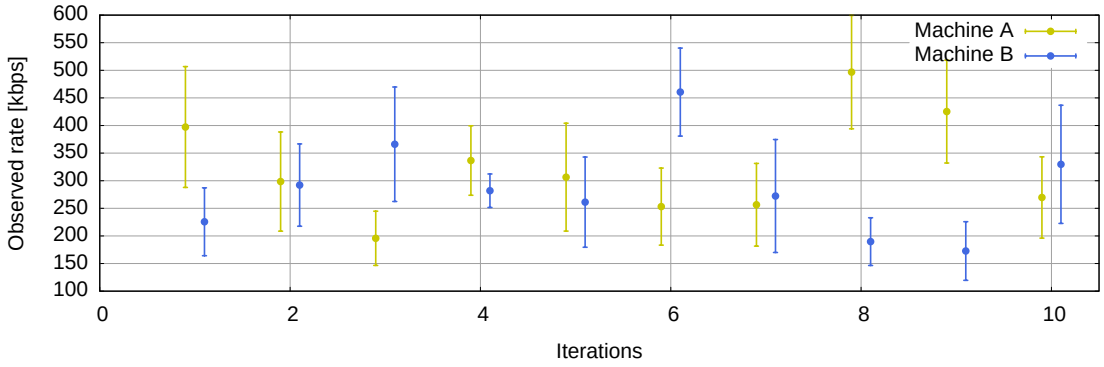
Studying the way the sender RRTCC takes decisions about rate adaptation we can observe that the available rate estimations calculated by the receiver are only reliable when the size of the queues along the channel is large enough [33] . When having short queues along the path, the maximum rate cannot be estimated without having packet loss on the link, as in this case the packet loss is negligible, the connection is not able to use the maximum amount of bandwidth available on the path.



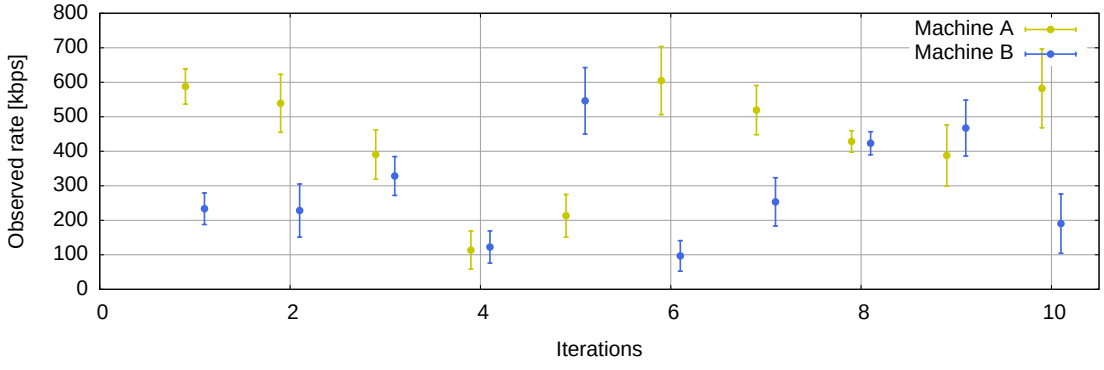
(a) 1 Mbit/s and 10s queue size.



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

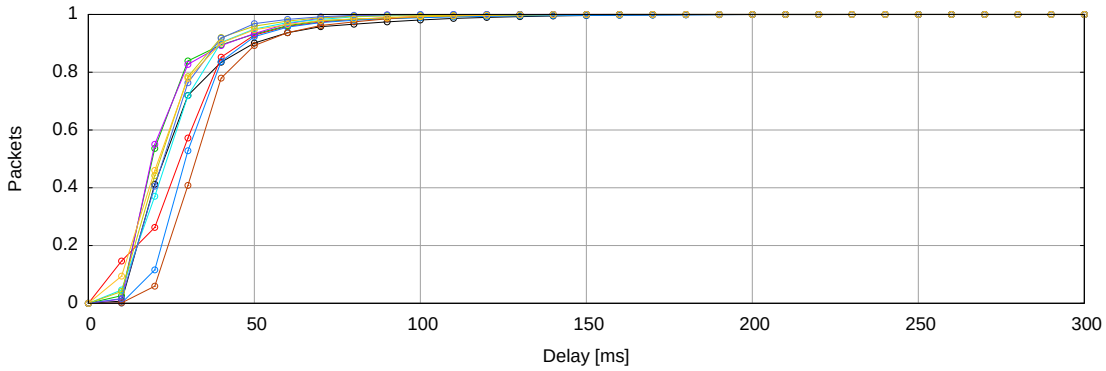


(c) 1 Mbit/s and 500ms queue size.

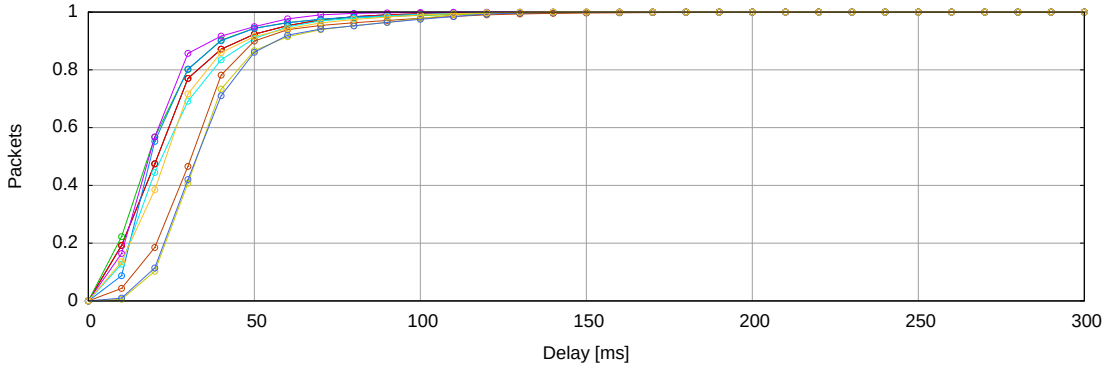


(d) 1 Mbit/s and 100ms queue size.

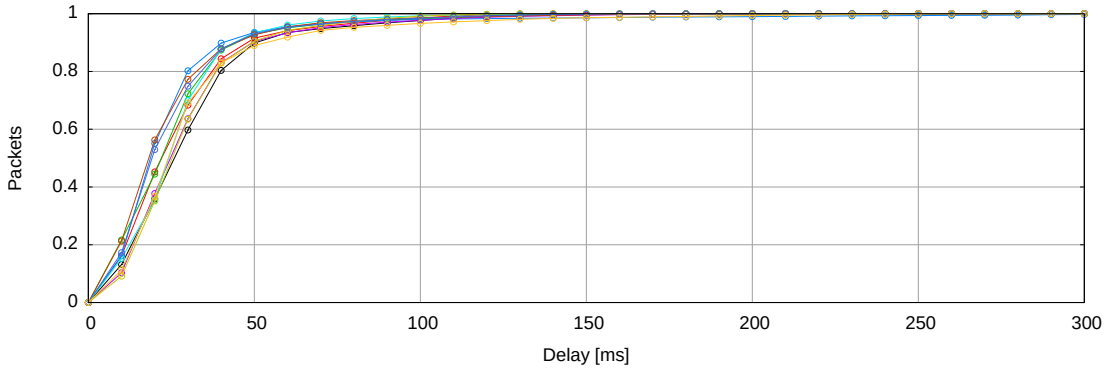
Figure 29: Bandwidth and mean for 1 Mbit/s with multiple queue sizes



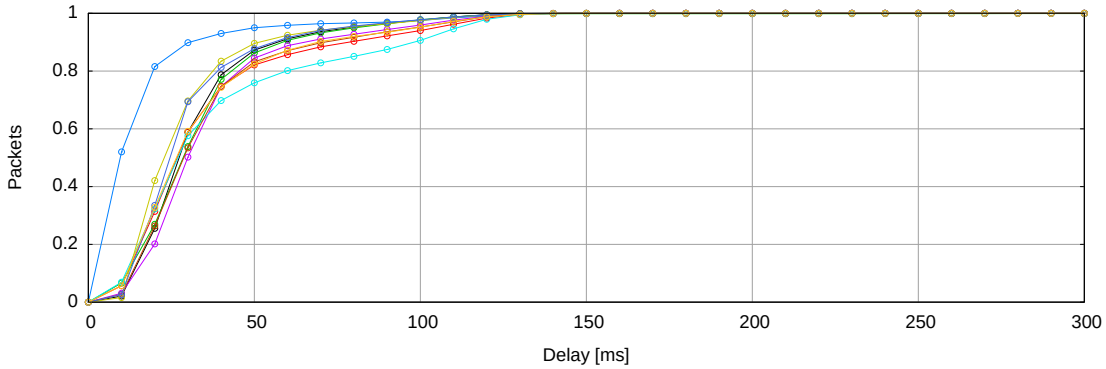
(a) 1 Mbit/s and 10s queue size.



(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: CDF of delay distribution for 1 Mbit/s with multiple queue sizes

6.3 Effects of TCP Cross Traffic

In this scenario, we are measuring the performance of WebRTC RTP flows when competing with TCP cross-traffic on the bottleneck with different path constraints. We use *Iperf* on the TURN server to emulate the TCP cross-traffic. On the other hand, *Iperf* clients will run on the peers that are performing the call. We use long TCP flows that represent large files being downloaded by the peers. Those flows run in parallel with the RTP media sessions stablished in WebRTC between the same clients.

The scenario used is the one shown in Figure 31 with the clients running *Dummynet* instead of the relay to simulate the path conditions.

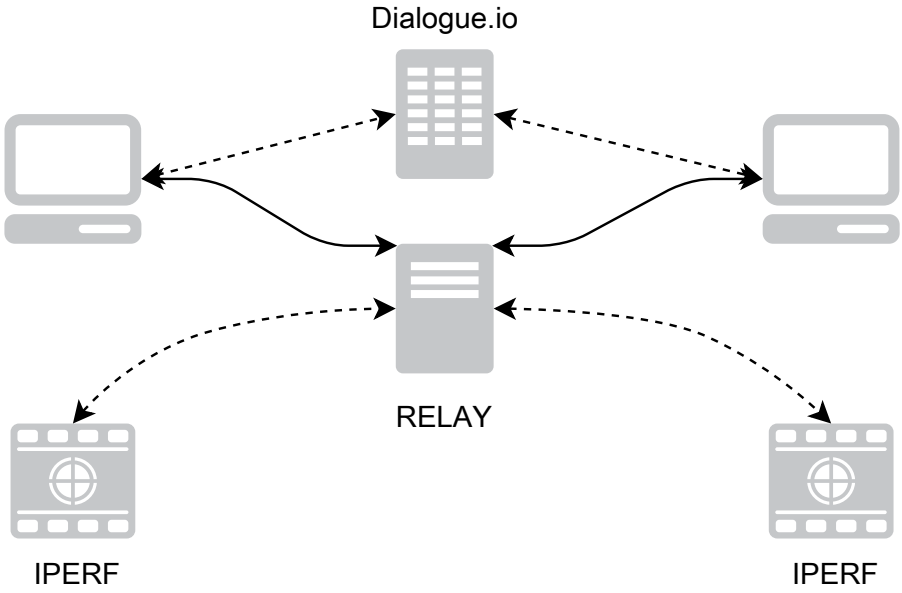


Figure 31: Topology for traffic flooded path using *Iperf*.

First we run the server as daemon on the TURN by executing:

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

In the next step, *Iperf* sends TCP packets, to do so we will run:

```
# iperf -c XXXX -t 300
```

In the previous command, *-t* is the amount of time the test length and *-c* is the feature that sets the process to operate as a client to the recipient address.

The first test is configured with a RRTCC flow competing with two TCP flows on the bottleneck. The capacity on the bottleneck is set to 1 Mbit/s upstream and 10 Mbit/s downstream. Considering the amount of data required by a RRTCC media stream, the 10 Mbit/s should be shared by the TCP and the remote stream. However, the 1 Mbit/s upstream could be used only with the local RTP media.

Table 9 shows that RRTCC starves to provide enough rate to the outgoing media and cannot compete with the TCP cross-traffic on the link, only a fraction of the bandwidth is given to the outgoing local stream and the receiver might not get proper quality on the incoming media.

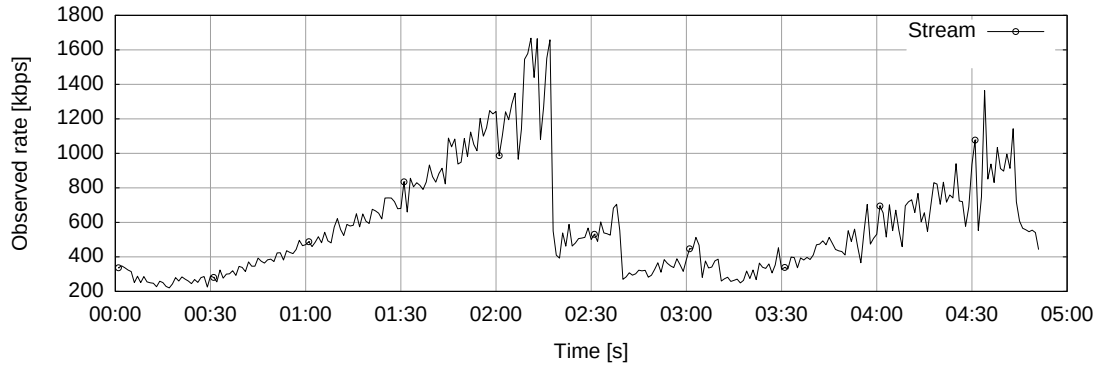
During the second test we set the bottleneck bandwidth to 10 Mbit/s upstream and downstream. The result, seen in Table 9, shows that RRTCC still starves trying to provide maximum rate to the outgoing media without succeeding.

10/1 Mbps				
Delay	Metrics	Machine A	Machine B	Overall
0 ms	**Rate (Kbit/s)**	76.16±25.12	74.81±24.93	75.49±25.02
	OWD (ms)	4.71±12.71	5.12±14.9	4.92±13.81
	Residual Loss (%)	0.23	0.44	0.26
	Packet Loss (%)	0.23	0.21	0.22
25 ms	**Rate (Kbit/s)**	79.97±25.7	88.07±32.65	84.02±29.17
	OWD (ms)	29.56±3.03	27.52±2.92	28.54±2.97
	Residual Loss (%)	0.13	0.38	0.24
	Loss (%)	0.13	0.25	0.19
100 s	**Rate (Kbit/s)**	78.2±27.62	82.2±33.57	80.2±30.6
	OWD (ms)	108.29±4.19	108.08±4.04	108.19±4.11
	Residual Loss (%)	0.24	0.38	0.19
	Packet Loss (%)	0.24	0.14	0.19
10/10 Mbps				
Delay	Metrics	Machine A	Machine B	Overall
50 ms	**Rate (Kbit/s)**	76.16±25.12	74.81±24.93	75.49±25.02
	OWD (ms)	4.71±12.71	5.12±14.9	4.92±13.81
	Residual Loss (%)	0.23	0.44	0.26
	Packet Loss (%)	0.23	0.21	0.22

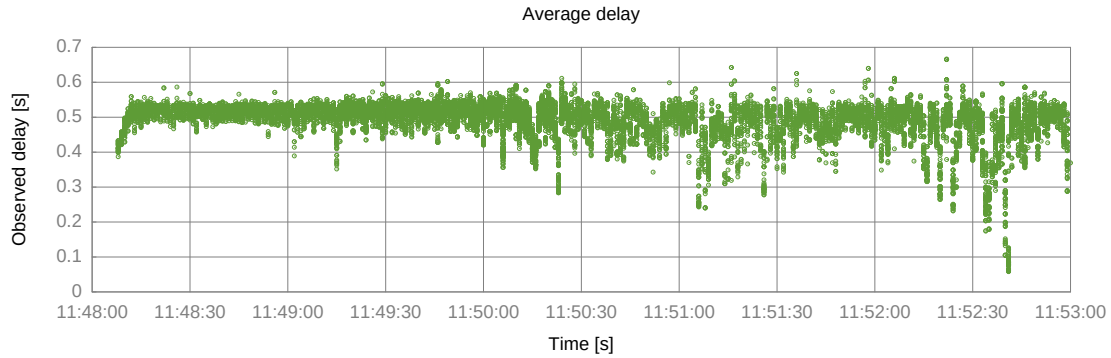
Table 9: Metrics for a bottleneck with varying amount of TCP cross-traffic and link constraints.

To analyze how RRTCC performs during a call we can observe Figure 32, both graphs represent the same stream in the duration of a call. During the whole period, TCP keeps on increasing its congestion window and, at the same time, filling the queues of the middle routers. This action produces delay on the path triggering a reaction in the RRTCC algorithm that reduces the rate on the ongoing stream. This is the reason to have the ramp in Figure 32a increasing slowly until timeouts appear, at that time RRTCC reduces the rate by half trying to adapt to the new conditions. We can also observe that the rate increase and delay decrease is related in Figures 32a and 32b.

The delay decrease seen in Figure 32b could be given by the routers queues exceeding the limit and dropping incoming packets, this causes the TCP to go into slow-start triggering RRTCC mechanisms to increasing the rate value.



(a) Instantaneous receiver rate.



(b) Instantaneous delay during the call.

Figure 32: Impact of TCP traffic on the performance of RRTCC during one call, bottleneck capacity set to 10/10 Mbit/s.

We can observe some interesting behavior in all three iterations when looking at Figure 33 CDF delay distribution, the response varies from all three tests being all of them bad, a lot of unexpected delays occur during the call. The delay deviation is small but the tolerance for TCP flooded networks is low in WebRTC.

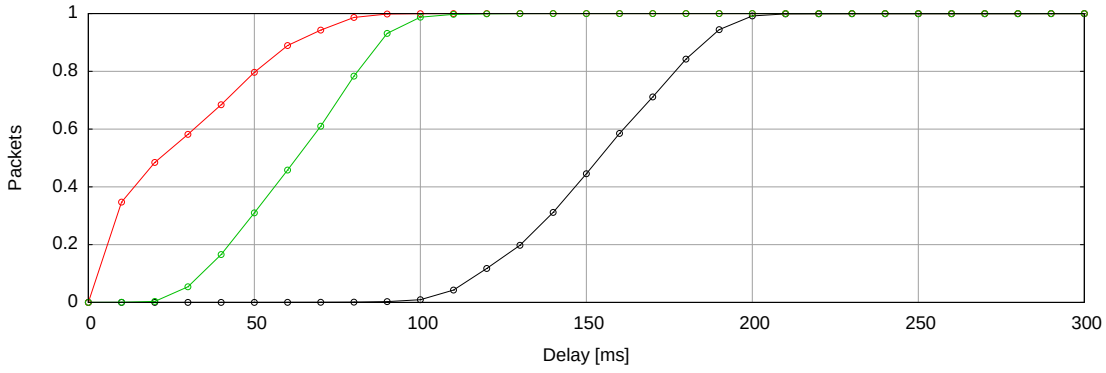


Figure 33: Total CDF delay distribution for TCP cross-traffic with 10/10 Mbps link condition.

6.4 Effects of RTP Cross Traffic

Similar to the previous section, in this chapter we are testing the performance of RRTCC when having multiple RTP flows on the same path through the bottleneck. Figure 34 represents the topology used for the test.

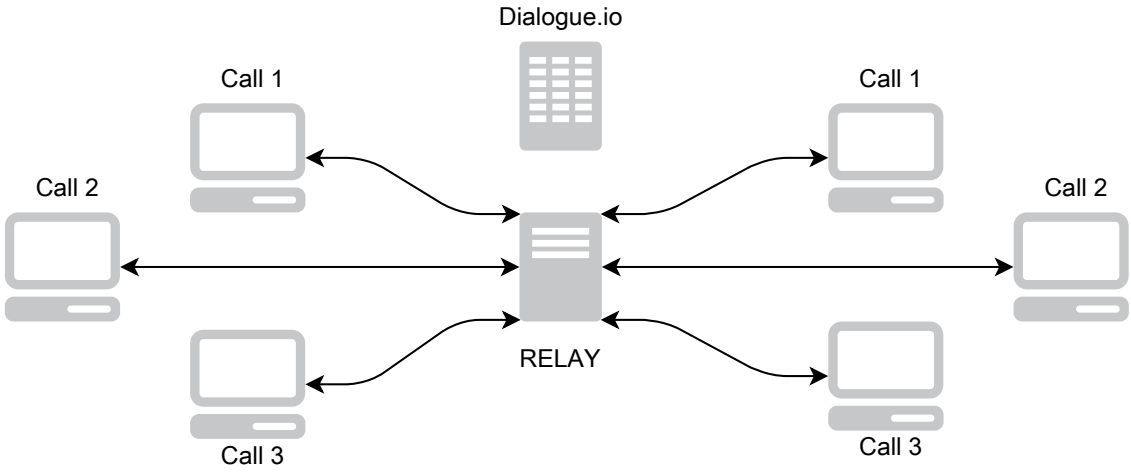


Figure 34: Topology for three different parallel calls using the same link.

Three different environments are tested, firstly two WebRTC calls share a common bottleneck. For this test we have two RTP streams running through the TURN server with different *Dummynet* constraints. We limit the bottleneck throughput with none, 10 or 20 Mbps. Table 10 shows the result for the test.

10/1 Mbps				
	Metrics	Machine A	Machine B	Overall
-	**Rate (Kbit/s)**	392.08±182.9	545.94±259.27	469.01±221.09
	OWD (ms)	5.1±4.8	5.9±5.69	5.5±5.26
	Residual Loss (%)	0.03	0.07	0.04
	Packet Loss (%)	0.03	0.04	0.03
10 Mbit/s	**Rate (Kbit/s)**	178.65±60.05	141.83±42.02	160.24±51.04
	OWD (ms)	8.34±9.78	8.18±9.67	8.26±9.72
	Residual Loss (%)	0.04	0.13	0.07
	Loss (%)	0.04	0.08	0.06
20 Mbit/s	**Rate (Kbit/s)**	432.56±141.31	531.13±169.82	481.85±155.56
	OWD (ms)	19.27±11.64	20.76±12.68	20.02±12.16
	Residual Loss (%)	0.65	1.06	0.85
	Packet Loss (%)	0.89	0.57	0.61

Table 10: Two parallel simultaneous calls with different bandwidth constraints on the link.

Observing Table 10 we can state that the calls are unable to occupy any sub-

stantial share of the bottleneck in any of the situations, being the difference between 20 and 10 Mbps barely negligible. They both struggle to obtain the necessary rate of 2 Mbps standard RRTCC maximum target.

Furthermore, the following test arrange three different calls that share the same bottleneck without any *Dummynet* constraint. The results seen in Table 11 shows that the average rate is higher but still not able to reach the maximum level of 2 Mbps.

	Machine A	Machine B	Overall
Rate (Kbit/s)	768.04±180.93	850.1±223.84	809.07±202.38
OWD (ms)	31.36±24.37	31.6±25.49	31.48±24.93
Residual Loss (%)	0.15	0.43	0.23
Packet Loss (%)	0.15	0.27	0.21

Table 11: Three simultaneous parallel calls on the same path without any link constraints.

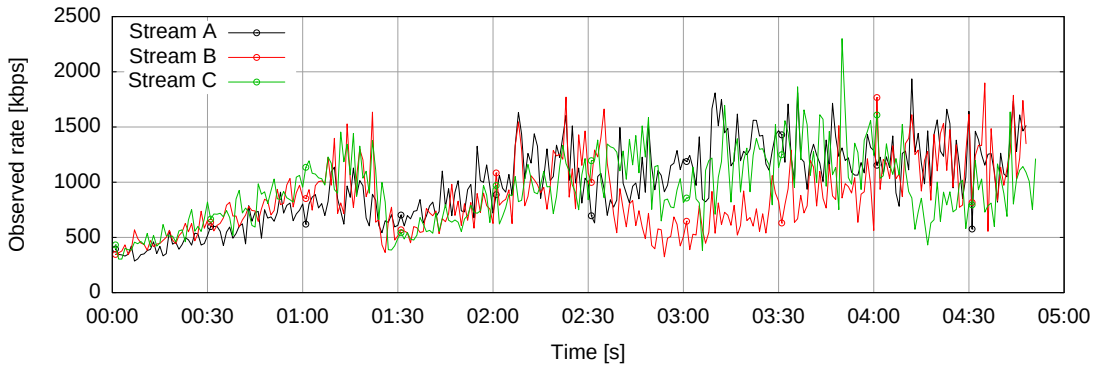


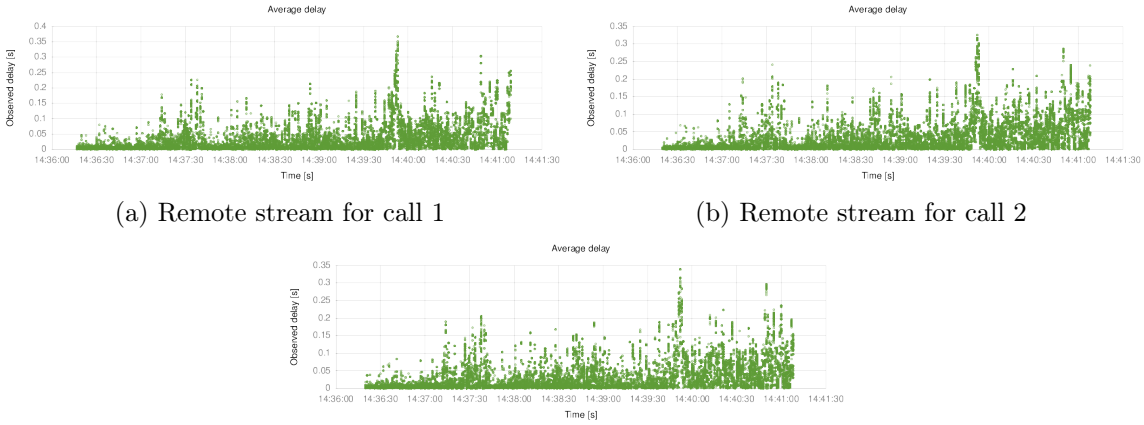
Figure 35: Variation in receiver rate for three parallel calls starting together.

Figure 35 shows three different RTP streams ramping up at a similar rate, reaching the similar peak and drop their rate together. All three calls start with different endpoints and using separate browsers, even dough, rates synchronize on the bottleneck as the RRTCC is acting in the same manner.

Figure 36 represents the delay on the same streams during the call, we can observe some peaks of delay during the same period as the rate decreases, the result of this is the sudden drop of rate.

In general the delay response in all the streams is bad, Figure 37 plots the delay distribution of the three simultaneous calls, the delay distribution produces variable unexpected delays, probably the user experience is not going to be optimal.

The last test is done by running the three calls starting at 30s intervals, by this we try to generate a more realistic approach to what could happen in a network. All three calls share the bottleneck link. Table 12 shows that the averaged rate for this test is slightly higher than the previous case. While the rate has improved compared



(a) Remote stream for call 1

(b) Remote stream for call 2

(c) Remote stream for call 3

Figure 36: Delay representation for all remote streams in a three peer parallel call.

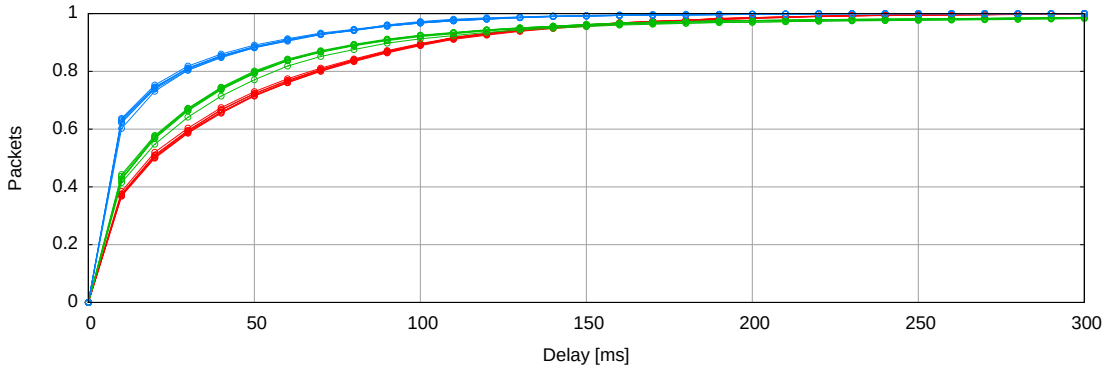


Figure 37: Total delay distribution for three parallel calls.

to the first test, Figure 38 shows that the first call has a disadvantage and, in all the cases, temporarily starves when new flows appear on the bottleneck, ramping up after a period of time.

	Machine A	Machine B	Overall
Rate (Kbit/s)	1214.66±247.41	1093.98±253.68	1154.32±250.54
OWD (ms)	34.86±27.09	35.44±28.68	35.15±27.88
Residual Loss (%)	0.08	0.17	0.91
Packet Loss (%)	0.08	0.09	0.08

Table 12: Three time shifted (30s) parallel calls on the same path without any link constraints.

Figure 38 shows the first call temporarily starving because it does not find any more flows on the link and the queues are empty. When the second stream appears, the existing session has to compete with the new one that observes some queues from the existing media. The reaction from the first flow is to reduce the sending

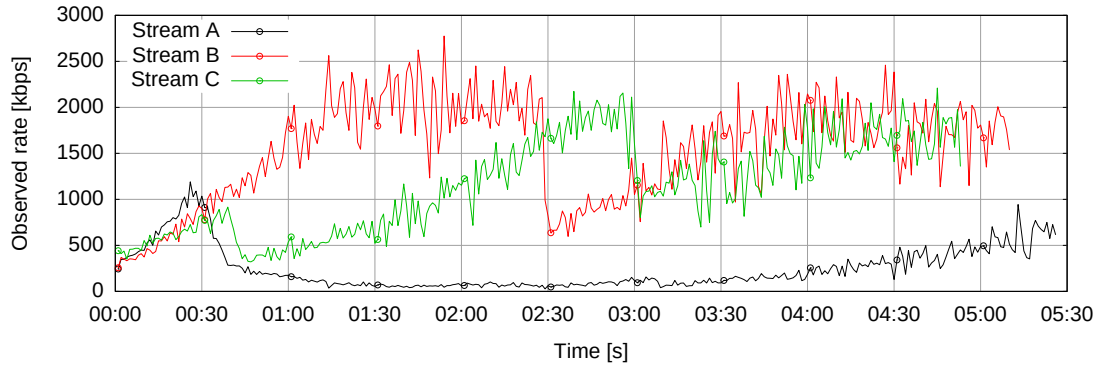


Figure 38: Variation in receiver rate for three parallel calls with 30s interval start time.

rate when it observes an increase at the queues in order to avoid congestion.

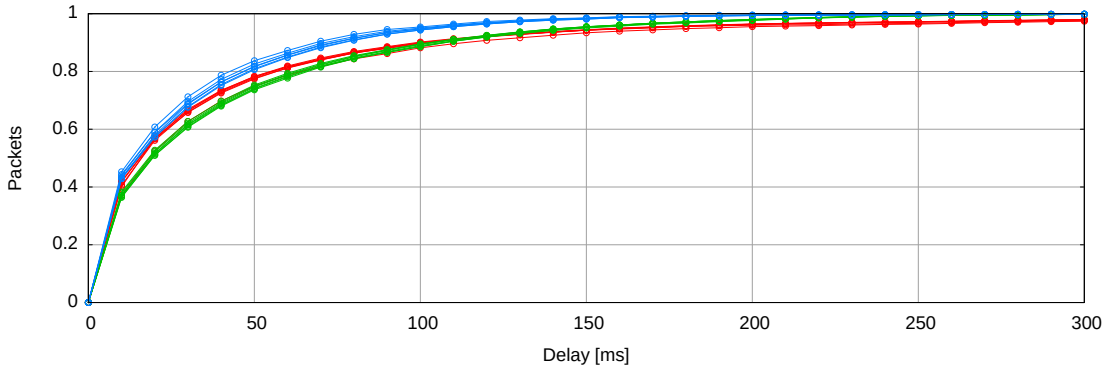


Figure 39: Total delay distribution for three asynchronous parallel calls.

Figure 39 represents the CDF delay distribution for the time shifted test, similar to the previous example (Figure 37) but with a worst delay response. Delay will affect to the user experience in the same way with more random variable delays.

6.5 Multiparty Calls: Full-Mesh

A common setup in real-time communications is video conferencing. We try to determine if RRTCC algorithm is able to cope with the requirements of a multiparty call. For this environment we setup a group call between three participants in a full-mesh topology. With this, we have each participant sending the media to the other two participants and also receiving the individual media from them.

Figure 40 shows a simple three-peer mesh topology like the one we are using in this test. In this scenario we do not use any TURN server or path constraint. The common bottleneck in an environment like this is the last and first hop in the path, this device has to handle lots of traffic and it's queues might be heavily loaded.

Table 13 shows the result for this setup. Observing the average rate seems obvious that the WebRTC stack cannot cope to deliver maximum rate to the any

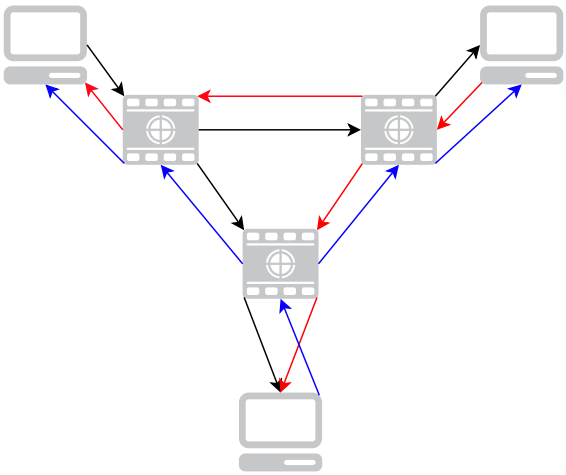


Figure 40: Mesh topology for WebRTC.

channel even without having path constraints. This might be given due to performance issues on the device itself. However, time response is good for real-time communication.

No MCU				
	Machine A	Machine B	Machine C	Overall
Rate (Kbit/s)	333.38±115.13	344.48±95.43	410.77±115.97	362.88±108.84
OWD (ms)	6.1±5.09	5.79±5.09	5.82±5.15	5.91±5.11
Residual Loss (%)	0.01	0.01	0.00	0.00
Packet Loss (%)	0.01	0.01	0.00	0.01

Table 13: Three-peer mesh call with and without TURN.

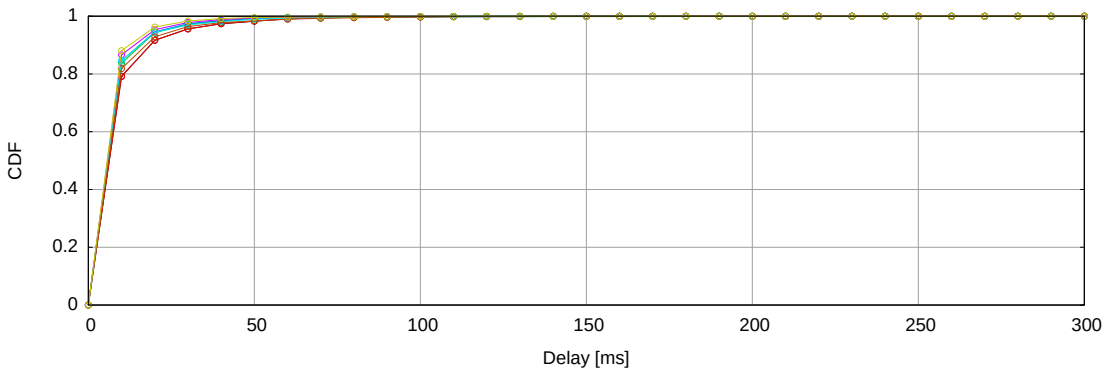


Figure 41: Total delay distribution for three peer mesh call without relay.

Figure 43 shows the receiver rate at each endpoint for a mesh call. We can notice that each endpoint runs as many congestion control mechanisms (RRTCC) as streams flows even the media encoding is done in the same stream multiple times.

Comparing with Table 13, we can also see that the rate deviation given on the results (Figure 42) is identified in Figure 43 with an slow ramp-up on the rate during the duration of the call.

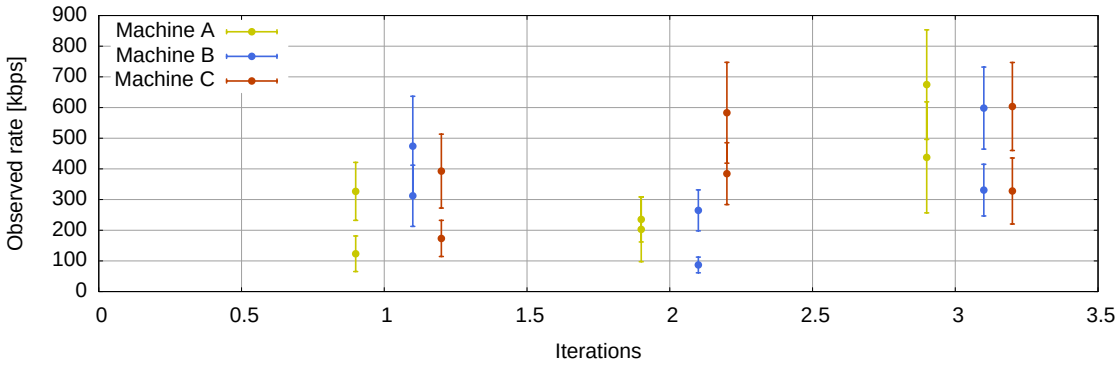
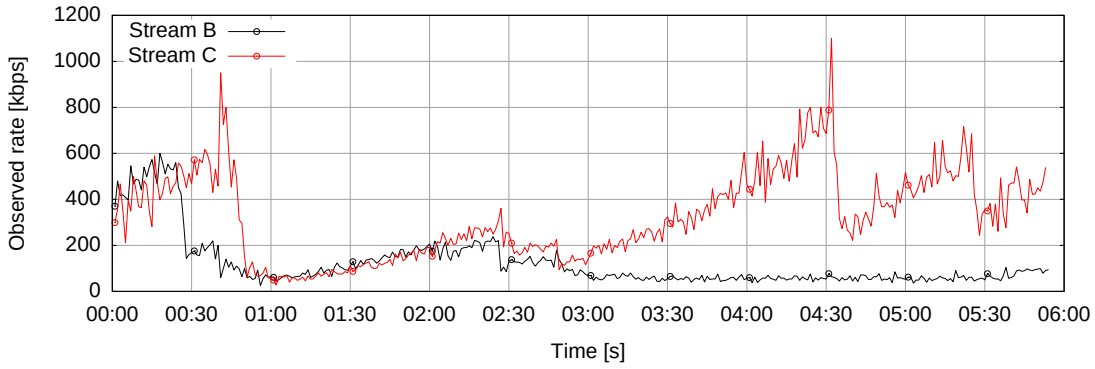
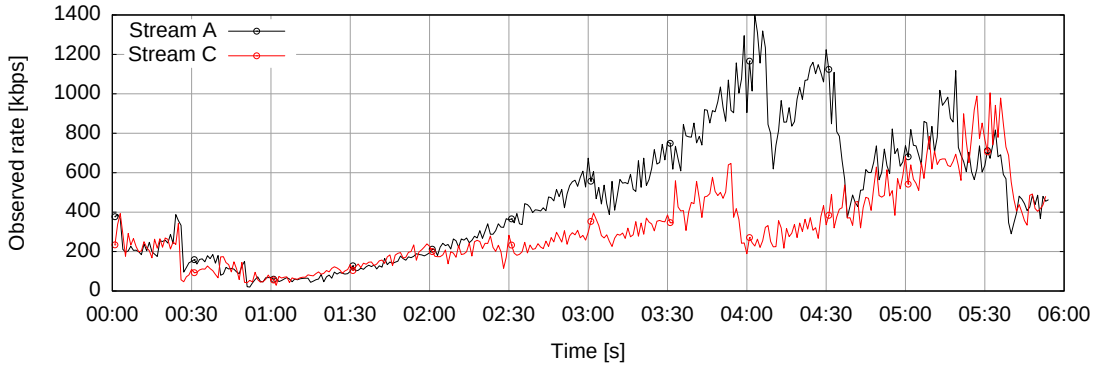


Figure 42: Bandwidth average and deviation for three peers mesh call.

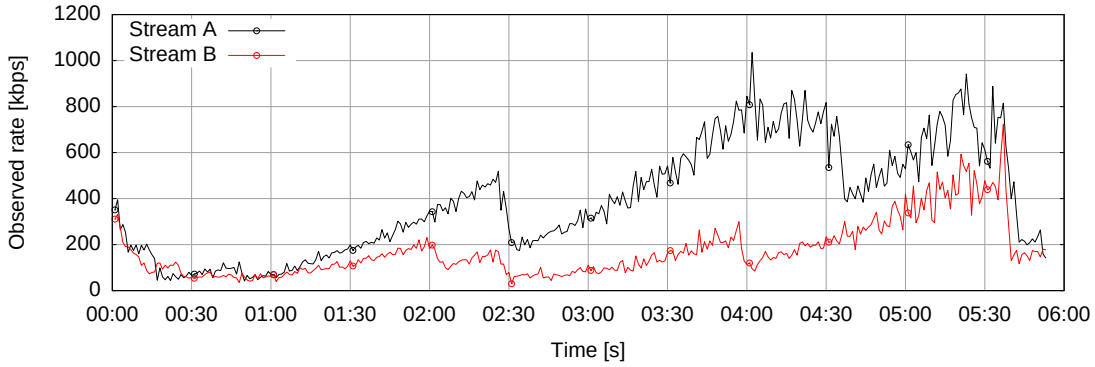
The CDF delay distribution response is good compared to previous scenarios, we can compare the obtained one in Figure 41 with the three parallel calls in Figure 37 and 39. Considering the curve of the delay for the mesh networks we can say that the delay won't be affecting significantly the user experience during the media session. This means that from the perspective of a non relayed mesh call we can have three peers with an acceptable rate and 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.



(a) At peer A



(b) At peer B



(c) At peer C

Figure 43: Variation of rate at the receiver endpoint for every incoming stream in a mesh call.

6.6 Multiparty Calls: Switching MCU

One common alternative for multiparty calls is the use of an MCU to perform the relying of the media through a unique device. 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. 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.

MCU				
	Machine A	Machine B	Machine C	Overall
Rate (Kbit/s)	604.31±149.38	403.74±93.99	882.94±228.45	630.33±157.27
OWD (ms)	6.88±3.94	6.31±3.8	6.4±3.64	6.54±3.8
Residual Loss (%)	0.05	0.06	0.07	0.02
Packet Loss (%)	0.05	0.06	0.07	0.06

Table 14: Three-peer mesh call with and without TURN.

We group a call of three peers using a centralized conferencing server operating a switching MCU, Figure 44 shows an example of a mesh topology using a MCU. This centralized server is a simple address translator and packet forwarder that receives the media and sends the tram to the individual remote participants. In this specific scenario, our MCU won't be generating the feedback packets for the RRTCC. Furthermore, the MCU collects the feedback packets from the endpoints and sends them back to the sender.

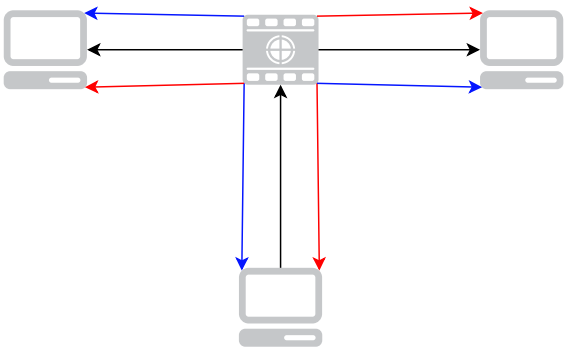


Figure 44: Mesh topology using a centralized MCU.

As result, even though an endpoint sends one media stream ,it receives back feedback reports for two participants. With this type of behavior the sender might get some conflict when taking decisions about the rate adaptation, as it might get two different reports stating to increase and decrease the rate. There is no specific solution for this issue in the RRTCC algorithm and it takes the best decision for each case.

Table 14 shows the results for the this scenario. We can see that the rate has doubled but the deviation has also increased.

Observing Figure 46 we can study the instantaneous rate at each endpoint. In some specific cases the link is unable to carry all the three media streams at a constant rate, as result one stream suffers from low rate. Stream C is suffering from this issue in Figure 46a and starves at both endpoints.

We can see a comparison between the previous scenario delay (no MCU) and the actual test in Figure 45.

Figure 45 represents the averaged delay result for the three iterations in both

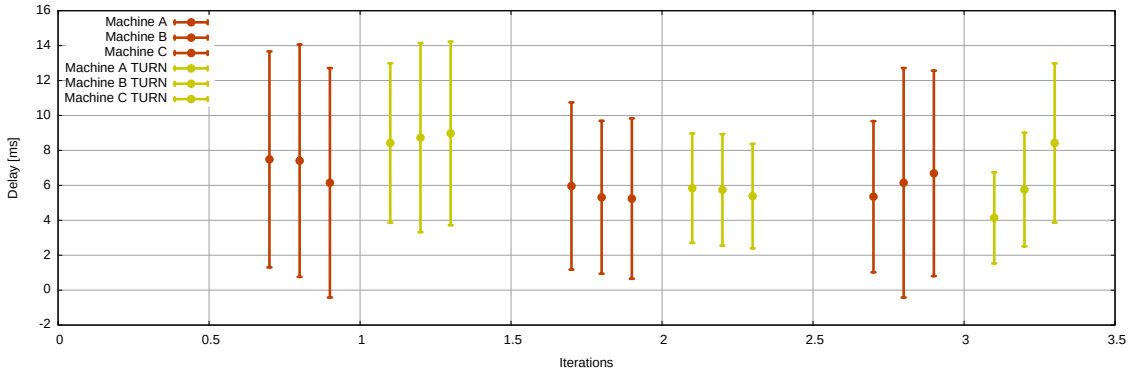
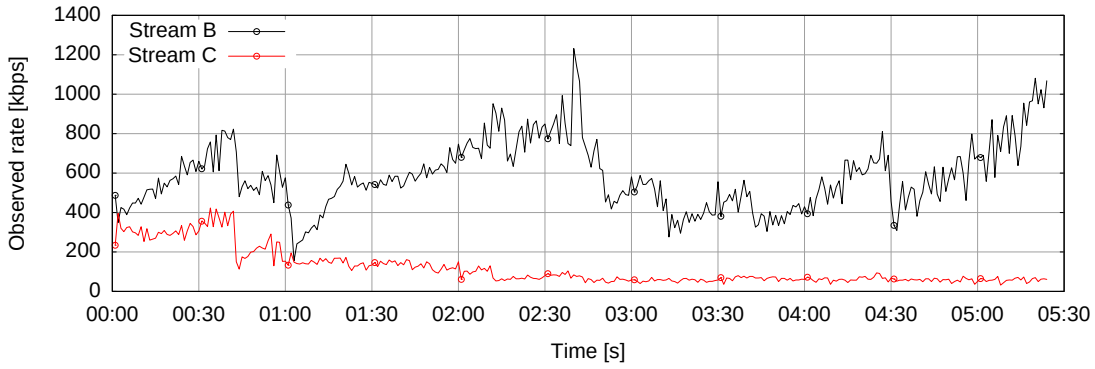
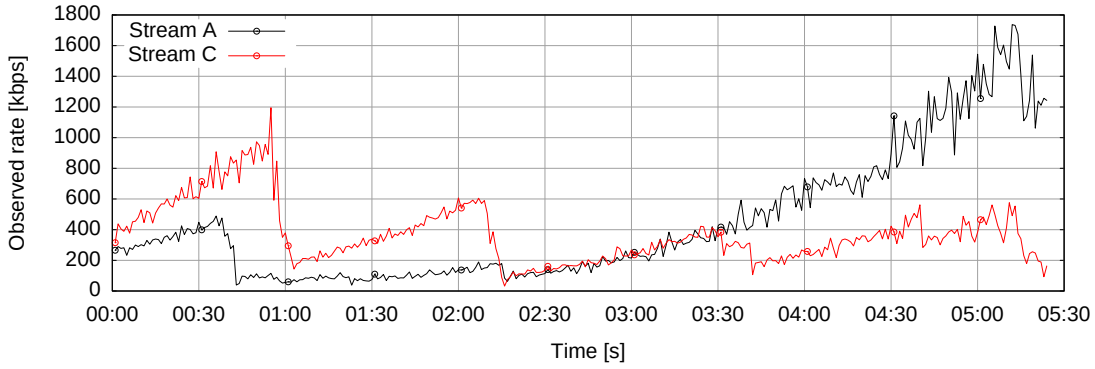


Figure 45: Averaged delay and deviation for TURN and non relayed mesh call for all iterations.

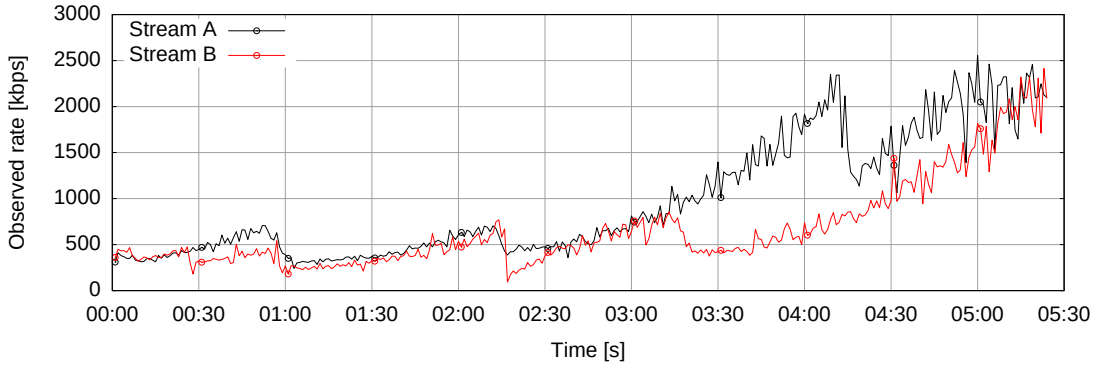
tests. Results are slightly better with the TURN, this might be due to the fixed path for routing the packets that produce smaller deviation, the averaged delay is similar in both scenarios.



(a) At peer A



(b) At peer B



(c) At peer C

Figure 46: Variation of rate at the receiver endpoint for every incoming stream in a mesh call with MCU.

6.7 Interoperability

One of the main goals of WebRTC is to provide interoperability between different browser vendors. In this chapter, we analyze if it is possible for all browsers to be interoperable with their implemented congestion control mechanisms.

Actually, three different browsers carry native implementations for WebRTC: Google Chrome, Mozilla Firefox and Opera. Of those, two of them include the *GetUserMedia* and *PeerConnection* APIs: Mozilla Firefox and Google Chrome, those

should be interoperable between them. Google Chrome has included WebRTC *PeerConnection* API for long time, but Mozilla Firefox delivered the *PeerConnection* API much later [18]. Making both browsers compatible required work from the vendors. Both engines should be able to understand their signaling messages and respond with the adequate SDP answer.

However, there are some differences in the SDP signaling messages, in the Firefox implementation, the implementation provides the STUN/TURN candidates bundled on the message meanwhile the API from Chrome sends the candidates separately by default, at the same time, the signaling messages from Chrome includes some extra information about the streams that is not given in the Firefox version [53].

Furthermore, both browsers must implement cross-compatible codecs to encode the stream and be able to handle the proper incoming RTP packets. Even though, most compatibility issues are solved, there are still some ongoing problems with the actual JavaScript method calls in both browsers [53].

One of the goals when checking the interoperability of the browsers, is to evaluate the congestion mechanisms already implemented in both internal engines. They should be able to manage the different environments in the a similar way. At the time of this document, not all congestion control mechanisms are included in Mozilla Firefox.

For this test we have executed a point-to-point call between a Firefox and Chrome in different machines.

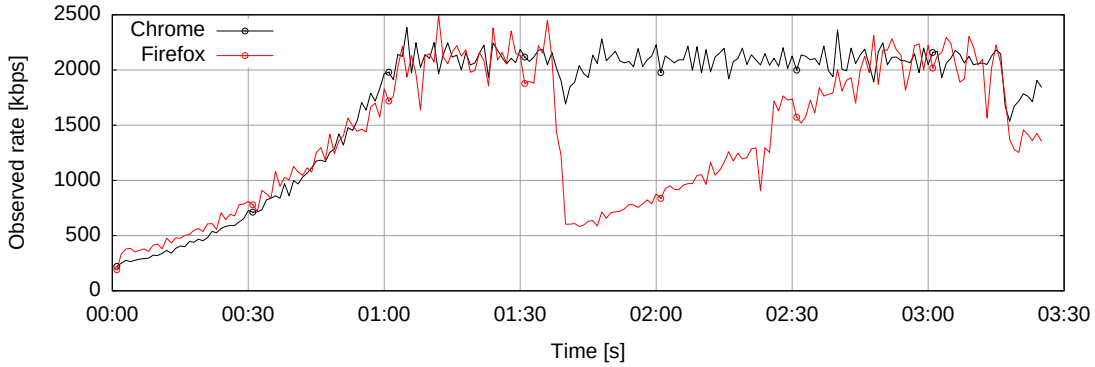


Figure 47: Remote stream rate for a point-to-point call between Mozilla Firefox and Google Chrome.

Figure 47 shows the given rate of the streams from both vendors captured at each receiver during the call. We can observe that congestion mechanism is not triggered in Firefox when some congestion occur on the path, Chrome applies rate adaptation meanwhile Firefox maintains the same stream rate over the link.

We should consider that Chrome RTP feedback messages use Receiver Estimated Maximum Bitrate (REMB) extra field for rate adaptation [33] [45]. This mechanism provides an extra field in the RTP feedback messages with the estimated total available bandwidth for the session.

To check the behavior of the RTCP feedback mechanism in WebRTC, we have captured different samples using a network sniffer. However, it is important to

advise that some of the congestion mechanisms, such as REMB, are not detectable by this software as it might be unable to decode non-standard fields.

Firstly, we obtained a capture from a call between two Chrome browsers, after analyzing the data, we can observe that both peers where exchanging *Sender Reports* control packets. In RRTCC, the RTT measurement of the RTCP packets is made by using the same RTP media flow timing, this avoids the usage of *Receiver Reports* in Chrome. We also notice that WebRTC multiplexes the RTP and RTCP packets in the same port to avoid extra usage of network resources [38].

Those messages carry important information required by the WebRTC internals to build the *Stats API*. The following fields (Listing 7) are extracted from a RTCP packet sent by Google Chrome during the test. We can see that most of the metrics are available in the packet, other metrics such as REMB are not able to be decoded [38].

Listing 7: RTCP message exchange between Chrome and Firefox

```
Real-time Transport Control Protocol (Sender Report)
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 0001 = Reception report count: 1
    Packet type: Sender Report (200)
    Length: 12 (52 bytes)
    Sender SSRC: 0xdf1a474d (3743041357)
    Timestamp, MSW: 3026830625 (0xb469c521)
    Timestamp, LSW: 653452974 (0x26f2e6ae)
    [MSW and LSW as NTP timestamp: Dec 1, 1995 18:17:05.152143000 UTC]
    RTP timestamp: 973093429
    Sender's packet count: 4236773172
    Sender's octet count: 3803919253
    Source 1
        Identifier: 0x9046a1ac (2420548012)
        SSRC contents
            Fraction lost: 102 / 256
            Cumulative number of packets lost: 4832630
        Extended highest sequence number received: 1896484047
            Sequence number cycles count: 28938
            Highest sequence number received: 3279
        Interarrival jitter: 1420789294
        Last SR timestamp: 2622976836 (0x9c577344)
        Delay since last SR timestamp: 2032955356 (31020436 milliseconds)

Real-time Transport Control Protocol (Receiver Report)
    10.. .... = Version: RFC 1889 Version (2)
    ..0. .... = Padding: False
    ...0 0001 = Reception report count: 1
    Packet type: Receiver Report (201)
    Length: 7 (32 bytes)
    Sender SSRC: 0xf46245fb (4100081147)
```

```
Source 1
    Identifier: 0x7f3343fb (2134066171)
    SSRC contents
        Fraction lost: 217 / 256
        Cumulative number of packets lost: 3472040
    Extended highest sequence number received: 4229724701
        Sequence number cycles count: 64540
        Highest sequence number received: 31261
    Interarrival jitter: 1181356458
    Last SR timestamp: 2930563385 (0xaeacd939)
    Delay since last SR timestamp: 3634988546 (55465523 milliseconds)
```

Furthermore, some other features must be answered by the WebRTC RTP engine, but might not be asked by the *PeerConnection* if they are not necessary or fully implemented. This is why features such as REMB may still not provided the RTCP *Sender Report*, if this field is not available, the congestion mechanism of Chrome calculates the estimated rate by using RRTCC mechanisms [33], those mechanisms use the information extracted from the RTCP report (7).

During this test with different vendors we have observed a different behavior in the RTCP mechanisms between Chrome and Firefox. Meanwhile Chrome continuously provide the *Sender Report* metrics in the RTP/RTCP channel, Firefox is only reporting *Receiver Reports* back to the source, this message exchange procedure is seen in the previous Listing 7. No information about local stream in Firefox is being sent to Chrome, this forces Chrome not to provide any feedback control messages to Firefox that would trigger their congestion control mechanisms. This behavior may affect the rate adaptation mechanisms in Firefox providing an output similar to Figure 47.

We can state that congestion mechanisms on Firefox are still not available and this avoids any rate adaptation in Firefox, in conclusion, using Firefox in multiple scenarios would lead to unexpected rate response and poor call quality. Besides from the congestion control mechanisms, Firefox and Chrome APIs are fully interoperable to perform calls.

6.8 Mobile Environment

In this section we test the response of RRTCC algorithm when using mobile networks, for this test we have used a mobile device carrying 3G connectivity that connects to a desktop device connected via ethernet cable. We can state that there was proper signal for the 3G environment.

Figure 48 represents the delivered rate in this scenario.

We can observe that the average rate for the cable stream is about 2 Mbps which is the maximum rate given by the RRTCC. However, the rate obtained in the 3G connectivity is approximately 200 Kbps.

Another important fact in this scenario is to observe the delay response. Figure 49 shows the delay produced in the 3G stream of Figure 48, we can observe large delay of over a second in some periods of the call. After the period of big delay there

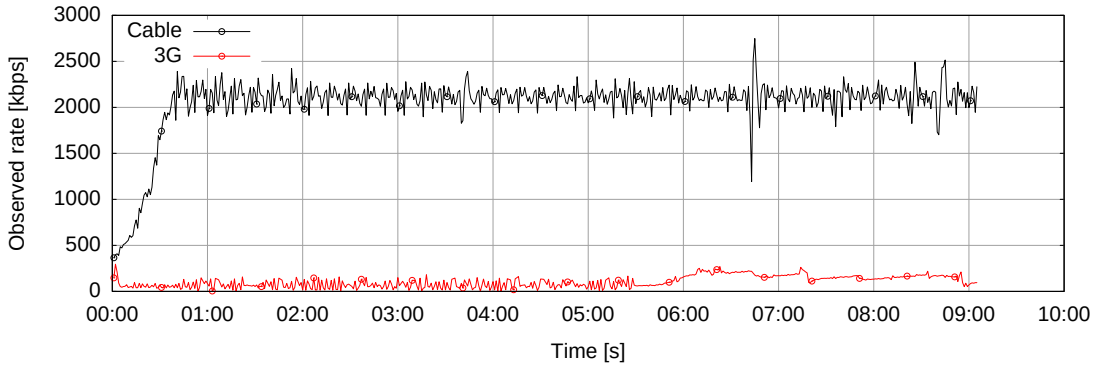


Figure 48: Rate obtained in mixed 3G and cable scenario.

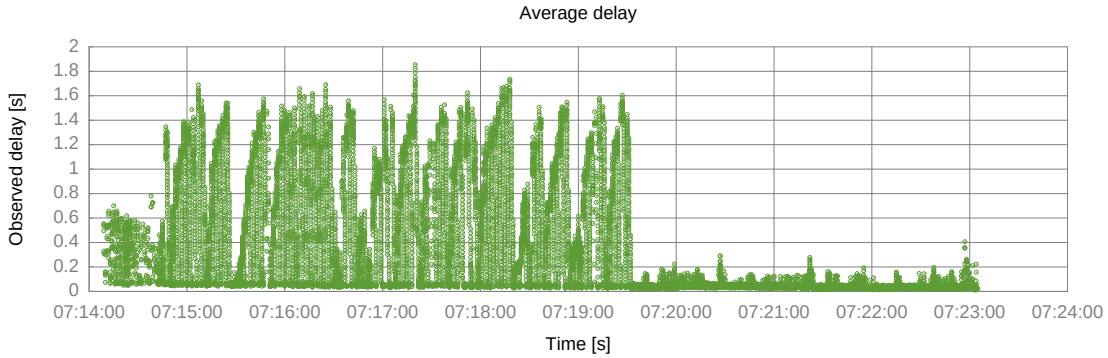


Figure 49: Delay response for the 3G stream in Figure 48.

is a sudden drop to 200ms. RRTCC is not able to cope with the delay produced when using 3G environment.

6.9 Resource Analysis

Lastly, we study which is the impact of WebRTC in terms of performance in a standard device. Figure 50 shows the averaged used resources in each different scenario. The only processes running during the tests have been the required tools for itself, no extra software is executed.

We can see that, in overall, WebRTC consumes a large amount of resources. Furthermore, when more than one stream is being handled the amount of CPU usage increases. As a consideration, the amount of CPU used in the three parallel calls test compared with the mesh might be studied. CPU usage in both scenarios is ≈90%, the consumption in the three parallel calls is still greater as there are three different local streams being accessed and played on the test, meanwhile in the case of mesh there is only one local stream captured and sent to the rest of peers.

When the CPU usage hits the maximum value, the quality of the call might be affected due to encoding limitations.

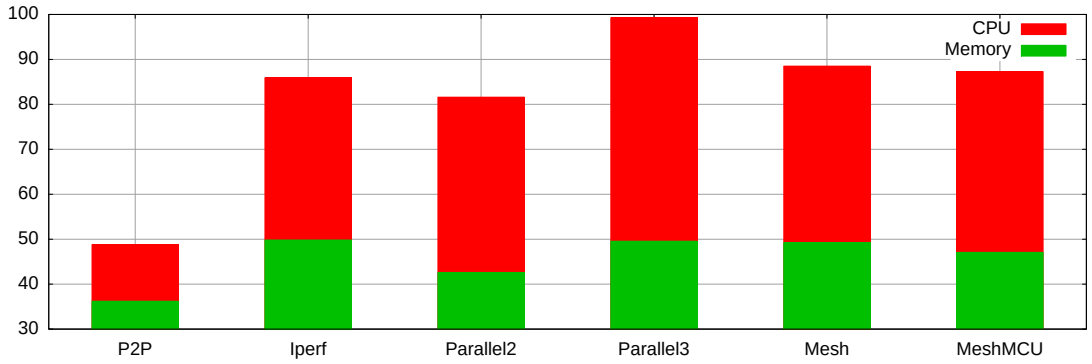


Figure 50: CPU and Memory usage in all the different executed tests.

6.10 Summary of results

After all the performed tests we can conclude that RRTCC algorithm implemented in Google Chrome performs well in low latencies networks up to 200ms but it collapses with greater values, this condition might be given in some specific environments such as mobile networks or long distance paths.

On the other hand, sending data rate under-utilizes the channel when competing with other TCP traffic on the same path, this is done to avoid an increase of latency. This increase of latency could provoke a rate reduction and directly affect the user experience.

RRTCC response improves when sharing the capacity with similar RTP flows. When having time-shifted flows, old streams can starve and state a low rate for a short period of time. This happens when multiple devices establish a time-shifted video call using the same bottleneck.

With mesh environments we can conclude that there is a under-utilization in the capacity of the path, this is given as a result of using independent congestion control engines for each flow. Each participant sends its local media in a lowered rate than expected.

We can summarize that RRTCC works well in low delay networks and can tolerate transient changes, competing correctly with some variable cross-traffic, within some limitation.

Meanwhile APIs are cross-compatible, we can state that WebRTC does not perform well when using different browser providers as the congestion control mechanisms haven't been fully implemented in all platforms.

Furthermore, superficial analysis suggest that RRTCC is still not ready to be used in mobile environments due the high delay produced by the network.

7 Conclusion

The end.

References

[1] 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.

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

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

[4] Salvatore Loreto. WebRTC: Real-Time Communication between Browsers. http://www.sloreto.com/slides/Aalto022013WebRTC/slides.html, 2013.

[5] The Future of Real-Time Video Communication, 2009.

[6] Jonathon N. Cummings and Brian Butler. The quality of online social relationships, 2000.

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

[8] Harald Alvestrand. Welcome to the list! http://lists.w3.org/Archives/Public/public-webrtc/2011Apr/0001.html, April 2011.

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

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

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

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

[13] J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with the Session Description Protocol (SDP). http://tools.ietf.org/html/rfc3264, June 2012.

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

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

[16] MS Open Tech publishes HTML5 Labs prototype of a Customizable, Ubiquitous Real Time Communication over the Web API proposal. http://msopentech.com/blog/2013/01/17/, January 2013.

[17] 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.

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

[19] 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.

[20] C.Holmberg, H. Alvestrand, and C. Jennings. Multiplexing Negotiation Using Session Description Protocol (SDP) Port Numbers. http://tools.ietf.org/html/draft-ietf-mmusic-sdp-bundle-negotiation, 2013.

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

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

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

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

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

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

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

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

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

[30] 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.

[31] J. Babiarz, K. Chan, and F. Baker. Configuration Guidelines for DiffServ Service Classes. http://tools.ietf.org/html/rfc4594, 2006.

[32] C. Perkins and V. Singh. Multimedia Congestion Control: Circuit Breakers for Unicast RTP Sessions. http://tools.ietf.org/html/draft-ietf-avtcore-rtp-circuit-breakers, 2013.

[33] 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.

[34] 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.

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

[36] A. Barth. The Web Origin Concept. http://tools.ietf.org/html/rfc6454, 2011.

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

[38] 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.

[39] 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.

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

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

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

[43] E. N. Gilbert. Capacity of a Burst-Noise Channel. *Bell Syst. Tech.*, 39(9):1253–1265, 1960.

[44] E. Elliot. Estimates of Error Rates for Codes on Burst-Noise Channels. *Bell Syst. Tech.*, pages 1977–1997, 1963.

[45] H. Alvestrand. RTCP message for Receiver Estimated Maximum Bitrate. http://tools.ietf.org/html/draft-alvestrand-rmcat-remb, 2013.

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

[47] Harinath Garudadri, Hyukjune Chung, Naveen Srinivasamurthy, and Phoom Sagetong. Rate Adaptation for Video Telephony in 3G Networks. *Proc. of PV*, 2007.

[48] V. Singh, J. Ott, and I. Curcio. Rate-control for Conversational Video Communication in Heterogeneous Networks. *Proc. of IEEE WoWMoM Workshop*, 2012.

[49] X. Zhu and R. Pan. NADA: A Unified Congestion Control Scheme for Real-Time Media. http://tools.ietf.org/html/draft-zhu-rmcat-nada, March 2013.

[50] P. O'Hanlon and K. Carlberg. Congestion control algorithm for lower latency and lower loss media transport. http://tools.ietf.org/html/draft-ohanlon-rmcat-dflow, April 2013.

[51] Xunqi Yu, James W. Modestino, and Dian Fan. Evaluation of the Residual Packet-Loss Rate Using Packet-Level FEC for Delay-Constrained Media Network Transport. http://info.iet.unipi.it/~luigi/papers/20091201-dummynet.pdf, November 2006.

[52] Marta Carbone and Luigi Rizzo. Dummynet Revisited, 2009.

[53] Interoperability notes between Chrome and Firefox. http://www.webrtc.org/interop, 2013.

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

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

[56] 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 [54].

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 [54] 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* [55], 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.[10]

```
# 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.

[10] 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 [52]. 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 [56], 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 [42].

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 8: 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 9: 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
```