Albert Abelló Lozano

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

School of Electrical Engineering

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

Espoo 20.3.2012

Thesis supervisor:

Prof. Jörg Ott

Thesis advisor:

M.Sc. (Tech.) Varun Singh

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Albert Abelló Lozano

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

Date: 20.3.2012 Language: English Number of pages:7+82

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 browser 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 priorly in other technologies are implemented in those browsers to handle the internals of WebRTC adding complexity to the system. 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

Abstract ii

Preface iii

Contents iv

1 Introduction 1
- 1.1 Background . . . 2
- 1.2 History . . . 2
- 1.3 Challenges . . . 3
- 1.4 Contribution . . . 3
- 1.5 Goals . . . 4
- 1.6 Structure . . . 4

2 Real-time Communication 5
- 2.1 Session Initiation Protocol (SIP) . . . 6
- 2.2 Real Time Media Flow Protocol (RTMFP) and Adobe Flash . . . 6
- 2.3 Web Real-Time Communication (WebRTC) . . . 8
  - 2.3.1 GetUserMedia API . . . 9
  - 2.3.2 PeerConnection API . . . 11
  - 2.3.3 Control and Monitoring . . . 12
  - 2.3.4 Low vs High level API . . . 13
  - 2.3.5 Internals of WebRTC . . . 15
  - 2.3.6 Security concerns . . . 17
- 2.4 Comparison between SIP, RTMFP and WebRTC . . . 19

3 Topologies for real-time multimedia communication 21
- 3.1 Point-to-Point . . . 21
- 3.2 One-to-Many . . . 22
- 3.3 Many-to-Many . . . 23
- 3.4 Multipoint Control Unit (MCU) . . . 23
- 3.5 Overlay . . . 24
  - 3.5.1 Hub-spoke . . . 24
  - 3.5.2 Tree . . . 25

4 Performance Metrics for WebRTC 26
- 4.1 Simple Feedback Loop . . . 26
- 4.2 Network metrics . . . 27
  - 4.2.1 Losses . . . 27
  - 4.2.2 Round-Trip Time (RTT) and One-Way Delay (OWD) . . . 27
  - 4.2.3 Throughput . . . 28
  - 4.2.4 Inter-Arrival Time (IAT) . . . 29
- 4.3 Host metrics . . . 30
  - 4.3.1 Resources . . . 30

4.3.2 Setup time . . . 30
4.3.3 Call failure . . . 30
4.3.4 Encoding and decoding . . . 30
4.4 Summary of metrics . . . 31

5 Evaluation Environment **32**
5.1 WebRTC client . . . 32
5.1.1 Connection Monitor . . . 32
5.1.2 Stats API . . . 33
5.1.3 Analysis of tools . . . 35
5.2 Automated testing . . . 35
5.3 TURN Server . . . 37
5.3.1 Dummynet . . . 39
5.4 Application Server . . . 39
5.5 Summary of tools . . . 39

6 Testing WebRTC **40**
6.1 Network Performance Evaluation . . . 40
6.1.1 Non-constrained link . . . 40
6.1.2 Lossy environments . . . 42
6.1.3 Delay varying networks . . . 44
6.1.4 Loss and delay . . . 46
6.1.5 Bandwidth and queue variations . . . 47
6.2 Loaded network . . . 52
6.3 Parallel calls . . . 56
6.4 Mesh topology . . . 60
6.5 Wireless scenario . . . 64
6.6 Interoperability . . . 65
6.7 Summary of results . . . 68

7 Conclusion **70**

References **71**

A Setting up fake devices in Google Chrome **75**

B Modifying Dummynet for bandwidth requirments **76**

C Scripts for testing WebRTC **79**

List of Figures

1 Real time communication between two users over the Internet . . . 5
2 SIP architecture for end-to-end signaling . . . 7
3 RTMFP architecture using Cirrus . . . 8
4 Market share of browser vendors by April 2013. Source [1] . . . 9

5 Media Stream API. Source [2] . . . 10
6 WebRTC simple topology for P2P communication . . . 11
7 JSEP signaling model . . . 14
8 WebRTC protocol stack. Source [3] . . . 16
9 Example of cross-site scripting attack . . . 18
10 WebRTC cross-domain call with Identity Provider authentication . . . 18
11 One-to-many topology for real time media . . . 22
12 Overlay topologies . . . 25
13 Multimedia feedback loop . . . 26
14 Description of simple testing environment topology for WebRTC . . . 32
15 Point-to-point WebRTC video stream throughput graph using ConMon over public WiFi at the network layer . . . 33
16 Point-to-point WebRTC video call total throughput graph using *Stats API* over public WiFi . . . 34
17 P2P video stream comparison between *ConMon* and *Stats API* . . . 35
18 Video stream rate with SSRC 0x646227 captured using *Stats API* and webcam input . . . 36
19 Video stream rate with SSRC 0x3a4df354 captured using *Stats API* and Chrome default fake content . . . 36
20 Video stream bandwidth using V4L2Loopback fake YUV file . . . 37
21 Rate average and deviation for non-constrained link . . . 40
22 Delay distribution for each P2P iterations with no link constraints . . . 41
23 Mean and deviation for OWD in each P2P iteration with no link constraints . . . 41
24 OWD response during the call for one video stream for a non-constrained link call . . . 42
25 Bandwidth mean and deviation for P2P 200 ms delay test . . . 45
26 Remote stream bandwidth for 10% packet loss rate and 50ms delay . . . 46
27 Remote stream bandwidth for 1 Mbit/s and 500ms queue size . . . 48
28 Stream delay for 1 Mbit/s and 500ms queue size . . . 48
29 Bandwidth and mean for 1 Mbit/s with multiple queue sizes . . . 50
30 Delay distribution for 1 Mbit/s with multiple queue sizes . . . 51
31 Topology for traffic flooded path using *Iperf* . . . 52
32 Bandwidth mean and deviation for 10 Mbit/s TCP *Iperf* test without link constraints . . . 53
33 Total delay distribution for 10 Mbit/s TCP *Iperf* test without link constraints . . . 54
34 10 Mbit/s UDP/TCP *Iperf* test with 100/10 link condition . . . 54
35 2 Mbit/s UDP and TCP *Iperf* test with 20/4 link condition . . . 55
36 Topology for three different parallel calls using the same link . . . 56
37 Bandwidth representation for all remote streams in a synchronous three peer parallel call for first iteration . . . 57
38 Delay representation for all remote streams in a three peer parallel call 58
39 Total delay distribution for three parallel calls . . . 58

40 Bandwidth representation for all remote streams in an asynchronous three peer parallel call for iteration one . . . 59
41 Total delay distribution for three asynchronous parallel calls . . . 59
42 Mesh topology for WebRTC . . . 60
43 Bandwidth average and deviation for three peers mesh call . . . 61
44 Bandwidth plot during all the call for the incoming streams of each peer for the first iteration . . . 62
45 Delay output for Figure 44a incoming streams . . . 62
46 Total delay distribution for three peer mesh call without relay . . . 63
47 Averaged delay and deviation for TURN and non relayed mesh call for all iterations . . . 63
48 Point-to-point video stream plot using StatsAPI and ConMon data over WiFi . . . 64
49 Delay calculated on the same stream captured using ConMon in both ends over WiFi . . . 65
50 Remote stream rate for a point-to-point call between Mozilla Firefox and Google Chrome . . . 66

List of Tables

1 Feature comparison between SIP, RTMFP and WebRTC . . . 19
2 P2P metrics output for WebRTC call with no link restriction . . . 41
3 Rate, OWD and loss averaged results for different packet loss constraints on the link . . . 43
4 Summary of averaged bandwidth with different delay conditions . . . 45
5 Averaged bandwidth with different delay conditions with 10% packet loss . . . 46
6 IPERF 10 Mbit/s TCP test without link constraints . . . 53
7 IPERF 10 Mbit/s TCP and UDP test with constrained 100/10 Mbit/s link . . . 54
8 IPERF 2 Mbit/s TCP and UDP test with constrained 20/4 Mbit/s link . . . 55
9 Memory and CPU consumption rates for parallel calls in different link conditions . . . 56
10 Bandwidth rates for parallel calls in different link conditions . . . 57
11 CPU, memory and bandwidth results for three peer mesh scenario without relay . . . 61

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. The existence of reliable, low cost and simple platforms for real-time communication is becoming an essential part for the future of consumer behavior, business structure and innovation.

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.

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 [4]. At the same time, this is changing traditions and habits of communication between people and transforming personal relationships [5]. 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. Thus, 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 a must have feature in nowadays computers. Those two enablers along with the increased demand for richer applications for the WWW are some of the reasons behind 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) and 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 Program-

ming 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. This is commonly seen as a call system over the web or video calling applications.

Web application APIs are defined into the HyperText Markup Language (HTML) version 5 and help developers to add features to their web applications with minimal effort using JavaScript functions. APIs 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 work combining two different coding languages, HTML and JavaScript, HTML is the de facto format for serving web applications and JavaScript is becoming the most popular scripting system for web clients to allow users to dynamically interact with the web application.

The actual version of WebRTC is a merge between different API that integrate with each other to provide flexible RTC in the browser. However, the final goal is to allow developers to create plugin-free real-time web applications that can be cross compatible with different browser vendors and operating systems.

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 [6], and the official mailing list started in April 2011 [7]. 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 the 27th of October 2011 [8]. During this first W3C draft, only media (audio and video) could be sent 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 also joined the IETF with a working group in May 2011 [9]. 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 covered the

definition of the communication model, session management, security, NAT traversal solution, media formats, codec agreement and data transport [10]. In June 2011 Google publicly released the source code of their WebRTC API implementation [11].

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 short experience in WebRTC environments sharing the available resources, we might find some lack of documentation or previous literature regarding congestion analysis compared with other technologies.

The aim to test and help to develop new protocols such as WebRTC is unfortunately accompanied by a lack of information that may affect some of the statements made in this thesis. Hopefully, this should not affect its development neither its 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 two different issues: technical problems and political decisions. Firstly, congestion mechanisms for RTC have always been complicated to implement due to the need of a fast response against path disturbances and link conditions. In 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.

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

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

Not sure about here

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 intercommunicate 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 describes 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. In the signaling part, codec agreement and NAT traversal methods are decided at the same time as all the multiple features that will be enabled in the new session, making it crucial to configure the media and data to be transmitted.

On the other hand, once signaling is done data starts to flow to the receiver, this stream may include media (audio or video) and different types of data (e.g binary, text, blobs, etc.). During the transmission we may require also some extra signaling messages to be exchanged in order to maintain the link or adapt the constraints to the actual network conditions, at the same time, features might change during the session. Those messages are used for Offer/Answer model, congestion control adaptation or NAT traversal issues.

Some RTC technologies and protocols are: telephony, mobile phone communication, radio, instant messaging (IM) , Voice over IP (VoIP) , Video and Voice over IP (VVoIP) , Internet Relay Chat (IRC) and videoconferencing.

All the previous ways of communication work in real time and rely in Figure 1

topology but with different technologies (e.g SIP, RTFMP and WebRTC). In this thesis we manly work with Internet RTC using media and data.

2.1 Session Initiation Protocol (SIP)

SIP allows communication between two different users with audio/video support in real-time. SIP final Request for Comments (RFC) was published in June 2004, this document describes the original functionalities and mechanisms of SIP [12].

From an overview perspective, SIP is an application-layer control protocol for multimedia sessions which can establish, maintain and terminate media sessions. During the development of the standard, different new functionalities were added such as conferencing and the possibility of adding/removing media from existing sessions.

This protocol works alongside with other existing technologies such as Real-time Transport Protocol (RTP) , Session Description Protocol (SDP) and Media Gateway Control Protocol (MEGACO) . SIP uses SDP Offer/Answer model for the session description or capabilities negotiation between the end-points and RTP for the media transport, all these technologies are widely used in other protocols and usually provide legacy for older devices and outdated versions. Meanwhile SIP can locate and deliver a message to a user, SDP can provide the required capability for the session establishment and RTP can transport the data. Something remarkable when using SIP is the ability to perform federated calls between different domains.

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

DNS provides the IP address for both proxy servers and allows the messages to be exchanged between both peers, SIP uses the following three-way handshake: INVITE, 200OK and ACK. Those messages carry the SDP data inside in an object format, when ray@upc.cat receives the INVITE message from bob@aalto.fi, with the different capabilities of the caller, builds the 200OK response carrying the response with the chosen capabilities or the failure to establish the call. The media transport is done using RTP and RTCP over User Datagram Protocol (UDP) [12].

SIP is a pure VoIP confederated technology that helped the community to develop multiple protocols that are used in other real-time P2P communication scenarios.

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.

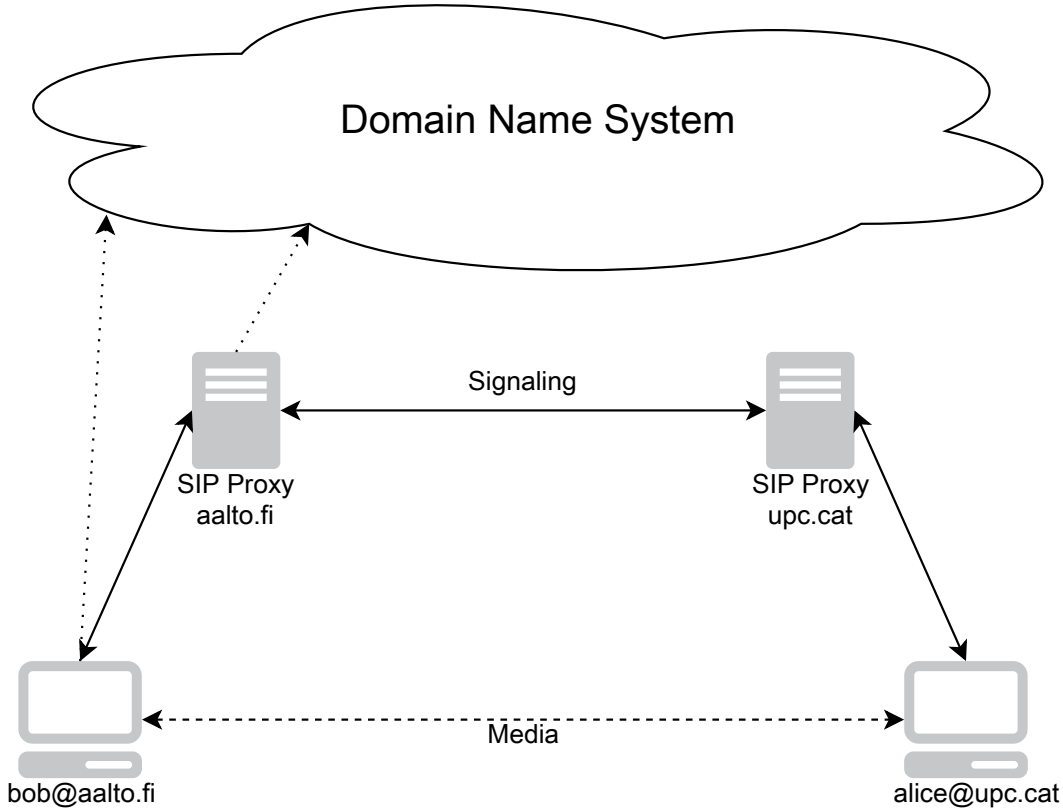


Figure 2: SIP architecture for end-to-end signaling.

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. Adobe Flash Player is available in most platforms, except iOS devices, and is present in about 98% of all internet-enabled desktop devices. This plugin allows developers to access media streams from external devices such as cameras and microphones to be used along with RTMFP.

This protocol is implemented by using Flash Player, Adobe Integrated Runtime (AIR) and Adobe Media Server (AMS) [13].

RTMFP uses Adobe Flash to provide media and data transfer between two end points over UDP [13]. RTMFP provides a full suite of methods and functions that allow the browser to access the necessary mechanisms to run real-time media communication, those methods are included into the plugin that must be installed prior usage. RTMFP is a private and licensed protocol. It also handles congestion

control on the packets and NAT transversal issues. One of the biggest differences is that, compared with SIP, RTMFP does not provide inter-domain connectivity and both peers must be in the same working domain to be able to communicate.

Media transfer is encrypted, this issue has been addressed in RTMFP by using proprietary algorithms and different encryption methods. The RTMFP architecture also allows reconnection in case of connectivity issues and works by multiplexing different media streams over the same media channel when handling conferences or multiple streams. For the signaling part, Adobe uses a service called Cirrus (Figure 3), this service allows architectures such as: end-to-end, many-to-many and multicast [14]. Those structures rely in the use of overlay techniques in multicast and mesh scenarios.

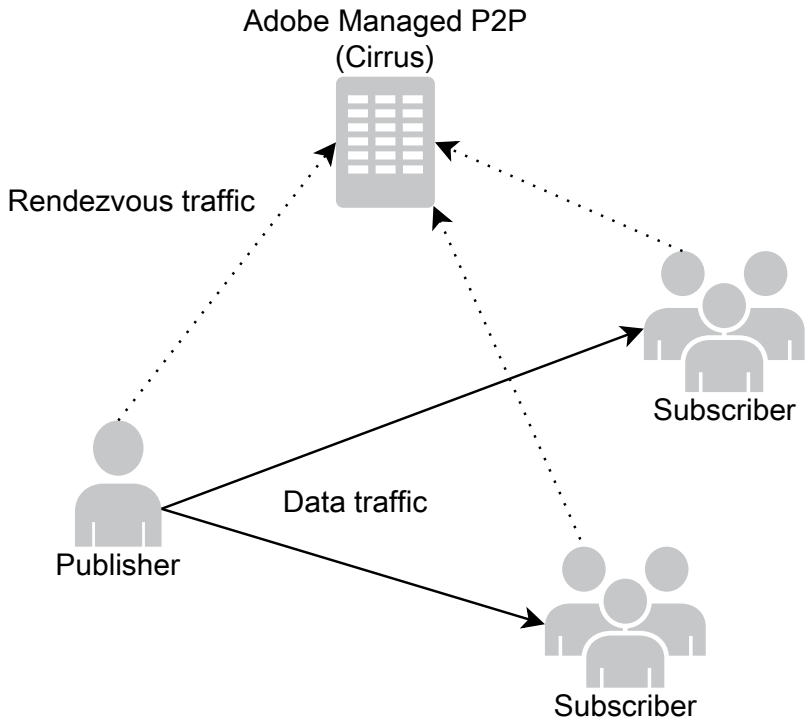


Figure 3: RTMFP architecture using Cirrus.

Some of the most valuable features is the possibility to easily 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 [6] [2], and enables RTC capabilities between Internet browsers using simple JavaScript APIs. Providing video, audio and data P2P without any plugins. This API replaces the need of installing a plugin to enable P2P communications between browsers, WebRTC uses existing standardized protocols, inherited from SIP, 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 actively implement WebRTC APIs are: Google Chrome, Mozilla Firefox and Opera. Other providers, such as Internet Explorer, are building prototypes for WebRTC [15].

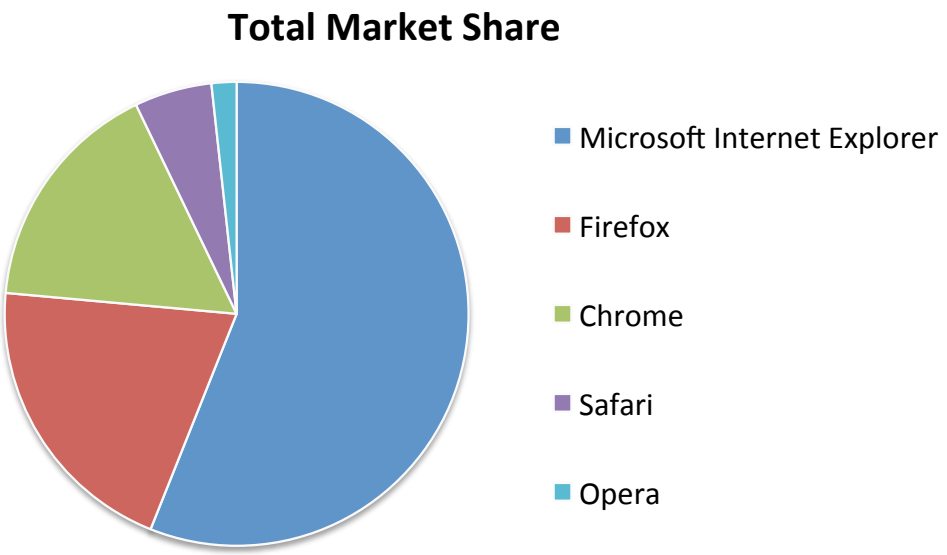


Figure 4: Market share of browser vendors by April 2013 [1].

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 [16].

WebRTC is composed by 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 uses *GetUserMedia* API to access 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*ÊAPI or with the HTML5 video element for playback purposes [2]. *GetUserMedia* is already interoperable between Google Chrome, Firefox and Opera [17].

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

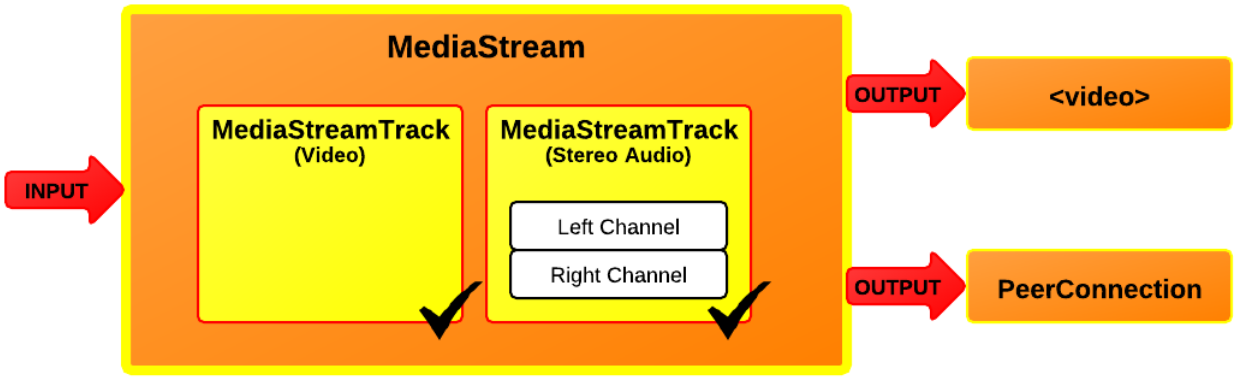


Figure 5: Media Stream API. Source [2].

for RTC video conferencing. The video tag is an HTML5 a 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 a close future, to include multiple sources of video and audio over the same stream from different devices. Other options include the design 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 in 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 the following code.

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);
}
```

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

GetUserMedia API also allow developers to set some specific constraints to the media acquisition. This help applications to better adapt the stream to their re-

quirements, those *cameraConstraints()* are stored into a JavaScript Object Notation (JSON) library and provided to the API through the *GetUserMedia* method.

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* [18]. *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.

Signaling messages are exchanged using a topology similar to Figure 6, 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.

WebRTC uses multiplexing over one port when sending the traffic, 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 [19]. *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. Consider that WebRTC works usually in a single domain and not in a federated environment as SIP.

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

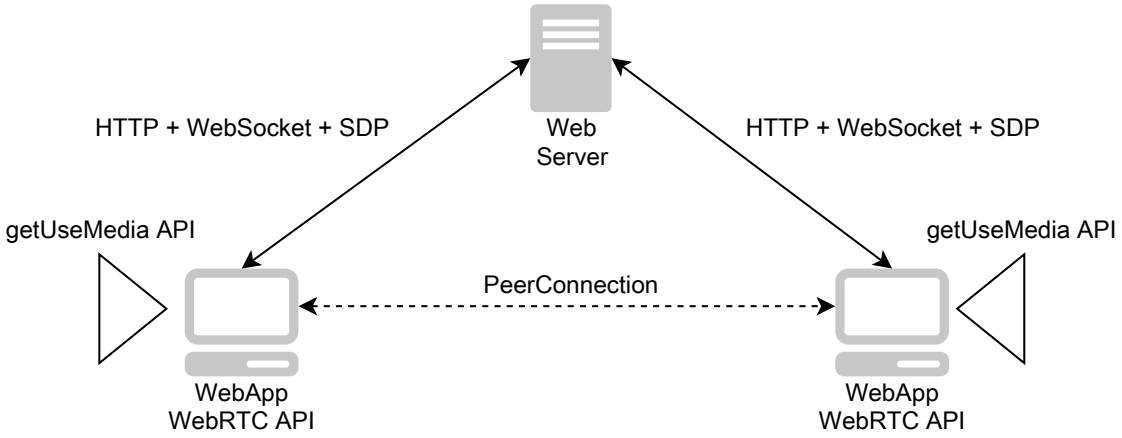


Figure 6: WebRTC simple topology for P2P communication.

Figure 6 does not show relay machines that help to provide NAT transversal solutions. In a real application cover this requirement using external services, those servers must be introduced into the WebRTC *PeerConnection* configuration when starting a new call.

Listing 2: Simple example of *PeerConnection* using JavaScript

```
//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);

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);
}
```

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

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 that may adapt the media constraints and configurations to the available resources on the link.

In WebRTC, this is done through the Statistics Model and Constraints defined in the W3C draft [18], these 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 in 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 [18], 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*.

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

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

Listing 3: JSON objects for constraints attributes in WebRTC

```
//Media constraints
var constraints = {
        "audio": true,
        "video": {
                "mandatory": {
                        "minWidth": "300",
                        "maxWidth": "640",
                        "minHeight": "200",
                        "maxHeight": "480",
                        "minFrameRate": "30"
                },
        "optional": []
        }
}

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

Both constraints objects are added to the *GetUserMedia* and *PeerConnection* methods when building the new 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 the

complexity level of the API, how much is available to be accessed by the developers and which configurations or mechanisms should be automatized in the browser. After long discussion, WebRTC is now using JavaScript Session Establishment Protocol (JSEP) [20], this API is a low level API that gives the developers control of the signaling plane allowing each application to be used in different environments, some applications give legacy to SIP or Jingle protocols meanwhile others might only work in a closed web domain.

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

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 that provides the different candidates that are required by the other peer in order to build the connection, those mechanisms are defined in chapter 2.3.5

One interesting feature that JSEP provides is called *rehydration*, this process is used whenever a page that contains an existing WebRTC session is reloaded keeping the existing session alive. This 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 [20].

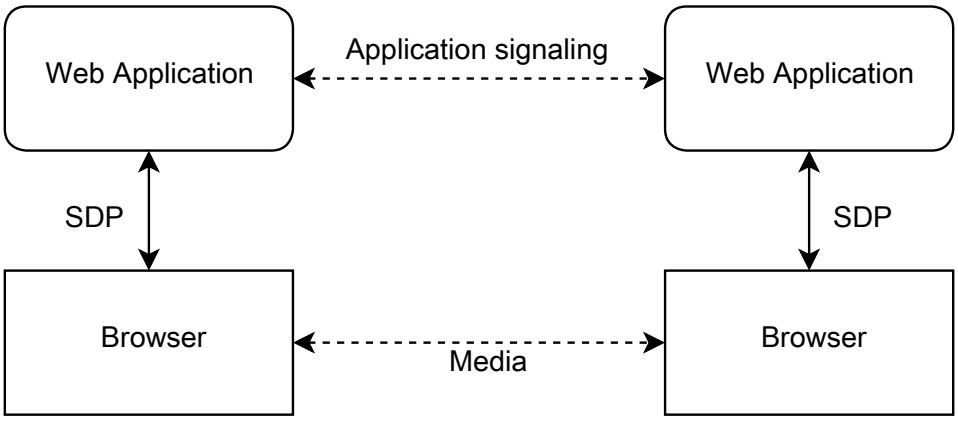


Figure 7: JSEP signaling model.

Low level APIs allow developers to build their own high level APIs that handle all the WebRTC protocol from media access to signaling. Those high level methods are useful to simplify the way JavaScript developers 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 are the multiple possibilities to adapt WebRTC to the requirements of each specific application.

We have developed our own high level API to create and handle RTC sessions and monitoring using WebRTC.

2.3.5 Internals of WebRTC

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

One of WebRTC main issues is NAT transversal, this problem usually affect 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 P2P media communications and has proven to be reliable when choosing the best option to establish communication [21].The enablers that work together with ICE for Real-time P2P protocols are Simple Transversal Utilities for NAT (STUN) and Traversal Using Relays around NAT (TURN) [22] [23].

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 provides mechanisms to discover the endpoints multiple IPs and ports that allow a direct connection to the peer behind the firewall, those interfaces are named *candidates*, this information is given to the sender that process the information and tries to choose the best *candidate*. On the other side, TURN works as a relay, this option should be always stated as the last resort when 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 except the case of TURN that might use TCP or UDP.

Media encoding in WebRTC is done through codecs implemented in the browser internals. Mandatory-to-Implement codecs for audio are G711 and Opus. G711 is an International Telecommunication Union (ITU) standard audio codec that has been used in multiple real time applications such as SIP. In real-time media applications, Opus is also a good alternative for G711, Opus is a lossy audio compression format codec developed by the IETF and that is designed to work in real-time media applications on the Internet [24]. 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 Canceler (AEC) and Noise Reduction (NR) . The first mechanism is a software based signal processing component that removes, in real time, the acoustic echo resulting from the voice being played out coming into the microphone (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 functioning 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 [25]. Non-media data in WebRTC is handled by System Control Transmission Protocol (SCTP) encapsulated over Datagram Transport Layer Security (DTLS) [26] [27] [25].

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. From the developer perspective, all the previous statements regarding security and transport are handled in the browser internals providing a simple and reliable way of sending P2P secure data over WebRTC.

WebRTC provides Secure Real-time Transport Protocol (SRTP) to allow media to be secured.The key-management for SRTP is provided by DTLS-SRTP which is an in-band keying and security parameter negotiation mechanism [3]. Figure 8 illustrates the full protocol stack for WebRTC described in this chapter.

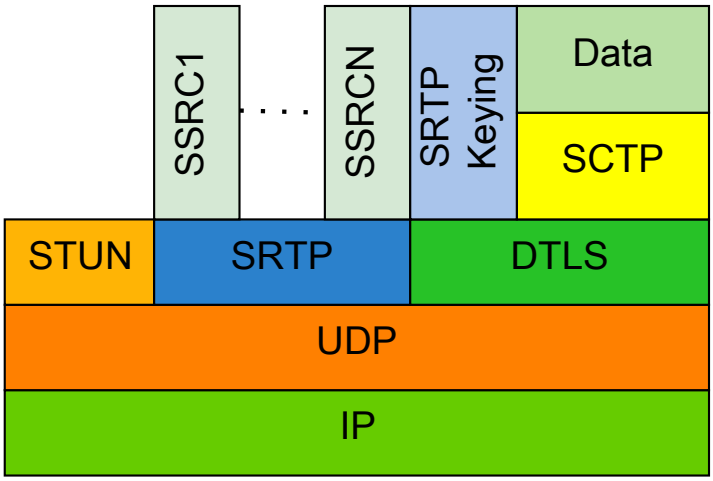


Figure 8: WebRTC protocol stack. Source [3].

Quality of Service (QoS) for WebRTC is also being discussed in the IETF and a draft is available with some proposals [28]. WebRTC uses DiffServ packet marking for QoS but this is not sufficient to help prevent congestion in some environments. When using DiffServ, the problem arises from the Internet Service Providers (ISPs) as they might be using their own packet marking with different DiffServ code-points, those packets are not interoperable between ISPs, there is an ongoing proposal to solve this problem by building consistent code-points [29]. Each specific application will mark Audio/video packets with the designed priority using DSCP mappings [28].

Officially, there is no congestion control mechanisms for WebRTC, the only mech-

anism actively used is circuit breakers for RTP [30]. Chrome specifically uses a Google congestion control algorithm that enables congestion control mechanisms for rate adaptation [31]. The aim of this algorithm is to provide performance and bandwidth sharing with other ongoing conferences and applications that share the same link.

2.3.6 Security concerns

To handle the signaling process in WebRTC we use a web server, peers exchange messages to 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 [32]. Figure 6 represents the simple topology for a WebRTC call, web 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. It has to avoid malicious calling when having a call established without user knowledge, considering that those APIs are able to bypass Firewalls and NAT, Denial of Services (DoS) attacks can also become a threat.

Actual browsers execute JavaScript scripts provided by the accessed web sites, this also includes malicious scripts, but in the case of WebRTC, this can produce 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 can execute a variety of actions in the browser. At a minimum, it should not be possible for arbitrary sites to initiate calls to arbitrary locations without user apprehension [33]. 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. Those situations, shown in Figure 9, are given when a third-party server provides JavaScript scripts to a different domain, this script cannot be trusted by the original domain that the user is accessing and could trigger browser actions that could harm the privacy. For example, in WebRTC, we could load a malicious script from a third-party entity that builds a WebRTC call to an undesired receiver without the user noticing this problem. Nowadays, browsers provide some degree of protection against XSS and do not let some scripting actions to be performed.

Other related vulnerabilities in WebRTC APIs is the possibility to establish media forwarding to a third peer, for example, once the user has accepted the access to the media, the provided JavaScript 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.

WebRTC calling procedure is done by the JavaScript provided by the server, this

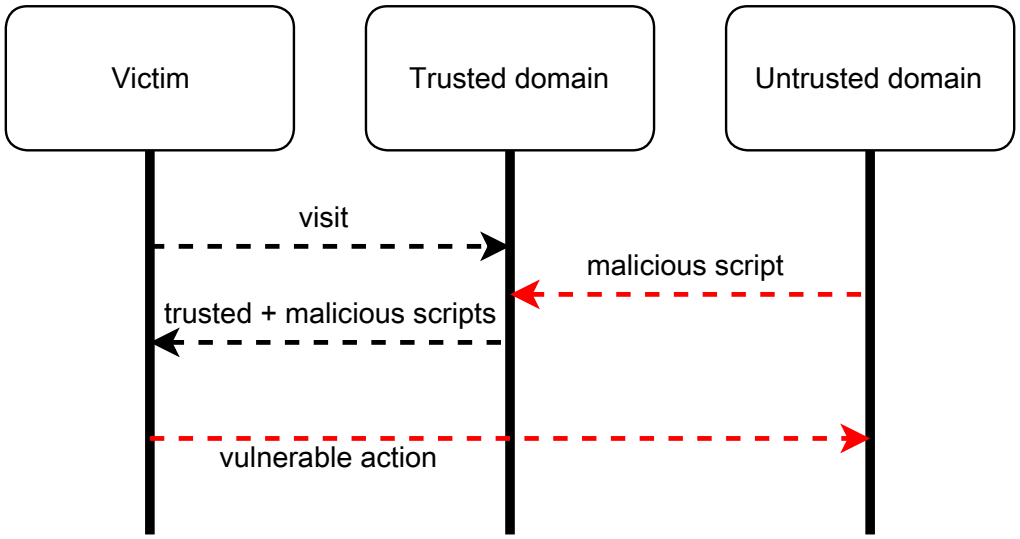


Figure 9: Example of cross-site scripting attack.

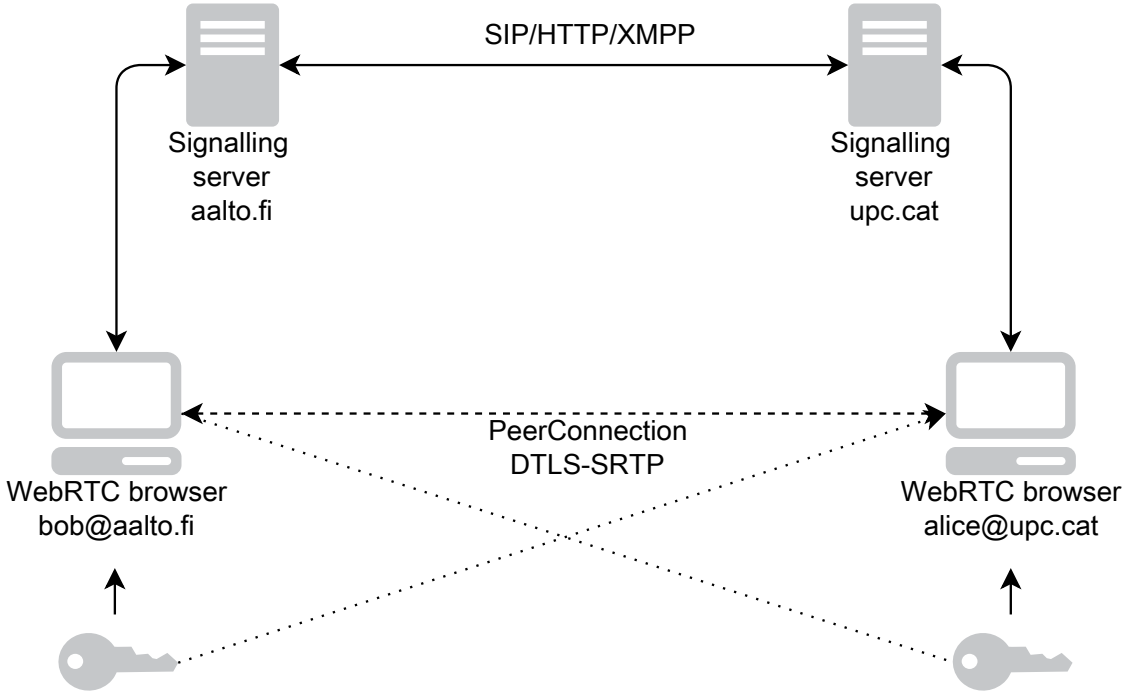


Figure 10: WebRTC cross-domain call with Identity Provider authentication.

may be a problem as the user must trust an unknown authority server. Calling services commonly use Hypertext Transfer Protocol Secure (HTTPS) for authentication whose origin can be verified and users 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 [34]. 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 10.

If the web service is running over a trusted HTTPS 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 actual API builds the ICE candidates for media verification. Authentication and verification in WebRTC is an ongoing discussion in the working groups.

Security an privacy issues in WebRTC can be given in multiple layers of the protocol, the increment of trust for the provider gives some vulnerability issues that sometimes cannot be easily solved if the aim is to keep a flexible and open 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 [32].

2.4 Comparison between SIP, RTMFP and WebRTC

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

	SIP	RTMFP	WebRTC
Plugin-enabled	No	Yes	No
Cross-domain	Yes	No	Maybe
Licensed	No	Yes	No
Android	Yes	Yes	Yes
iOS	Yes	No	Yes
Audio	Yes	Yes	Yes
Video	Yes	Yes	Yes
Data	No	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	No	No	Yes
SCTP	Yes	No	Yes

Table 1: Feature 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 the table to solve some of the issues.

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

From the mobile perspective, SIP is used in mobile technology and WebRTC has announced to be compatible with future versions of iOS and Android [16]. Furthermore, RTMFP has active support for Android but is still not able to extend its usage to iOS platforms.

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

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

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 11, 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.

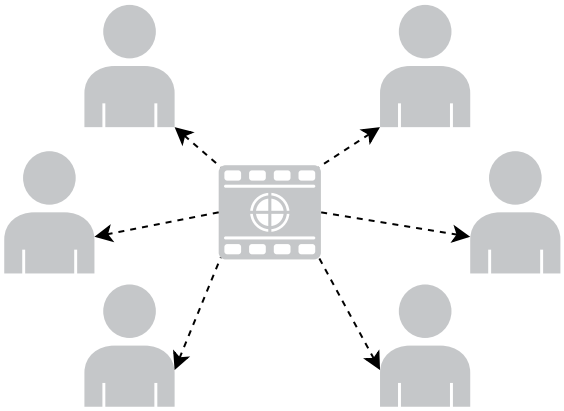


Figure 11: One-to-many topology for real time media.

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 12.

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

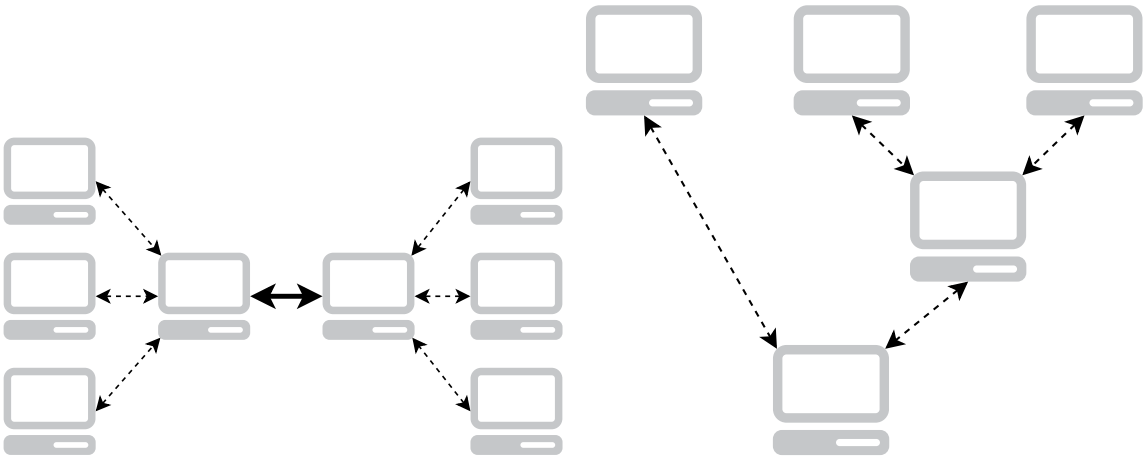
WebRTC does not provide native support for media overlay yet, but it is planned to implement those features in future versions of the API. Traditionally overlay has also been used for media streaming over the internet.

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 12a, 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 12: 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 12b.

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.

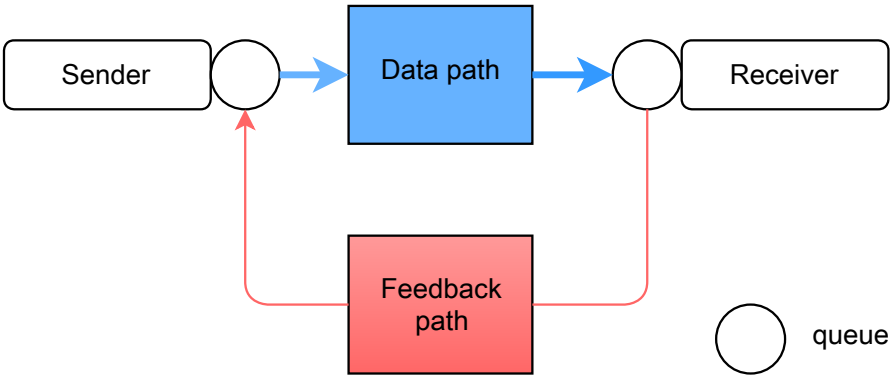


Figure 13: Multimedia feedback loop.

Figure 13 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 [35] [31].

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 [36]. 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 [36]. 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 [31].

In WebRTC, IAT is given by the Equation 6, 7 and 8 [31].

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

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

Our evaluation environment runs tests on WebRTC. Figure 14 describes the functional blocks used for a simple video call over WebRTC.

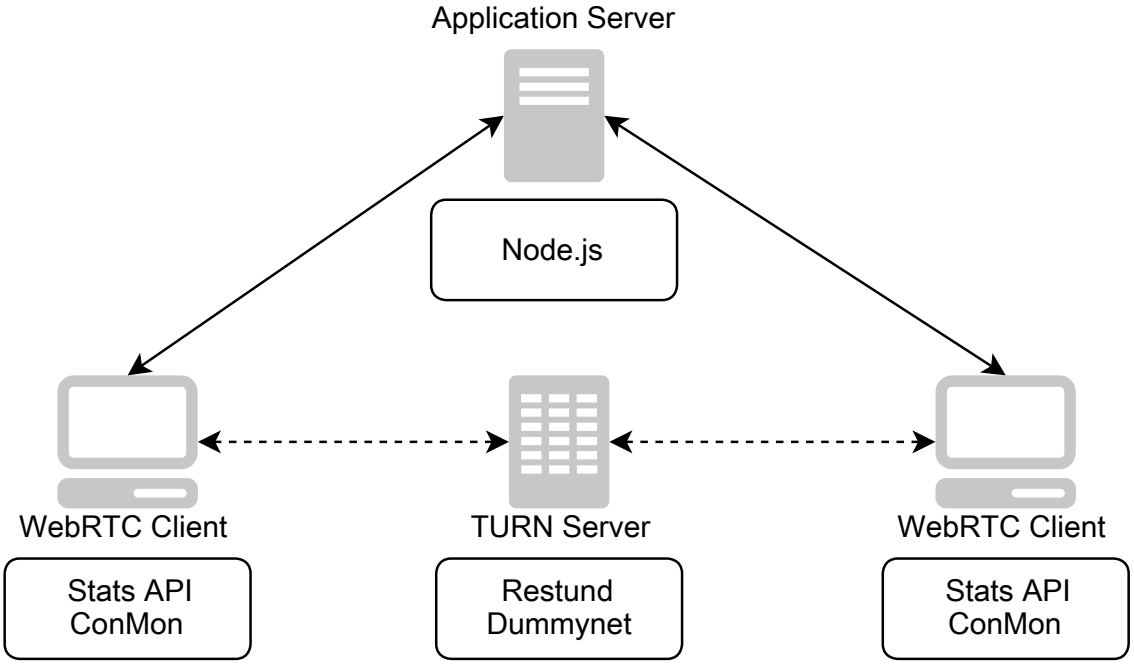


Figure 14: Description of simple testing environment topology for WebRTC.

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.

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 [37]. This utility is designed specifically to detect and capture RTP/UDP

[1]https://wiki.ubuntu.com/Lubuntu
[2]https://bugs.launchpad.net/ubuntu/+source/pulseaudio/+bug/1170313
[3]http://www.tcpdump.org/

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 15 represents one video stream from the same call as Figure 16 captured from the *ConMon* application.

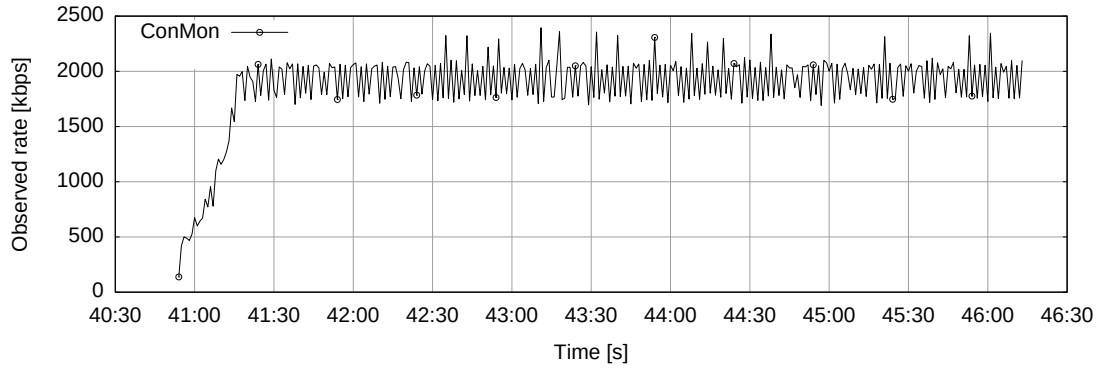


Figure 15: Point-to-point WebRTC video stream throughput graph using ConMon over public WiFi at the network layer.

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.

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 differ-

ences 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 [18] 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 [35].

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 [18].

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 16 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].

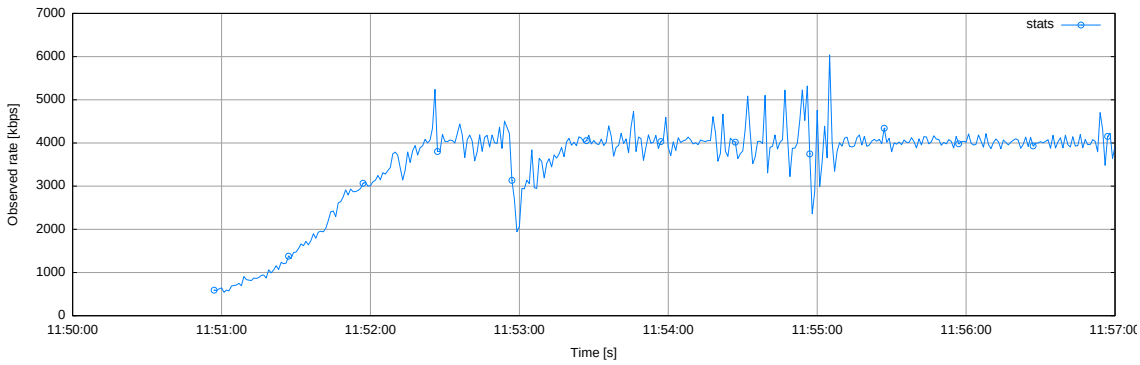


Figure 16: Point-to-point WebRTC video call total throughput graph using *Stats API* over public WiFi.

Figure 16 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

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

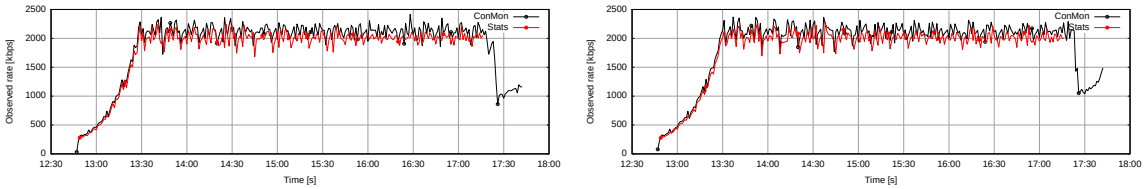
plot all the different streams together to get an idea of how much bandwidth the *PeerConnection* is consuming.

5.1.3 Analysis of tools

Stats API and *ConMon* measure the same metrics but from different layers of the 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 45.

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 17a and 17b plot two video streams being captured from Stats API and *ConMon*.



(a) Incomming stream.

(b) Outgoing stream.

Figure 17: P2P video stream comparison between *ConMon* and *Stats API*.

Figure 17 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.

5.2 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 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.

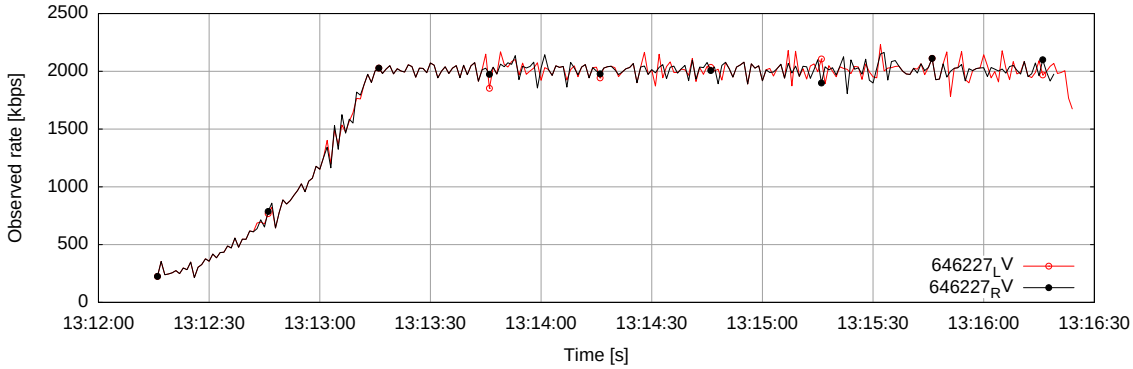


Figure 18: Video stream rate with SSRC 0x646227 captured using *Stats API* and webcam input.

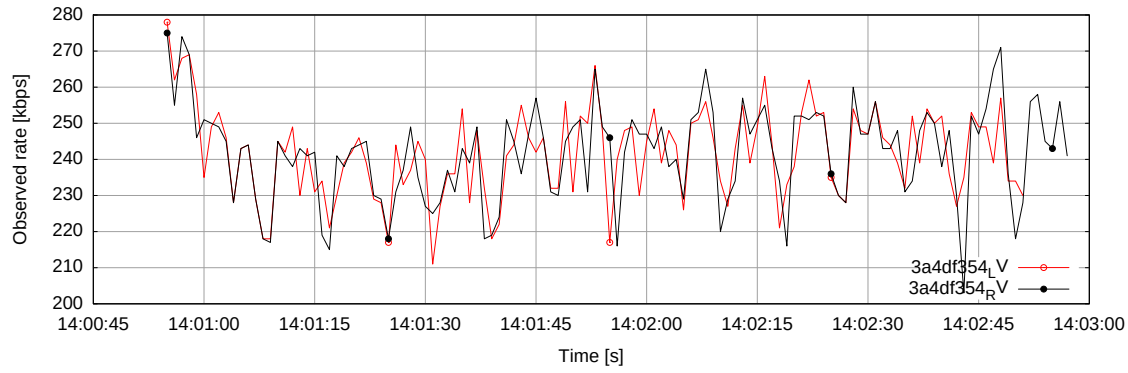


Figure 19: Video stream rate with SSRC 0x3a4df354 captured using *Stats API* and Chrome default fake video input.

Figure 18 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 19 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 (18 and 19) 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.

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

Comparing global output from Figures 18 and 19, 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 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.

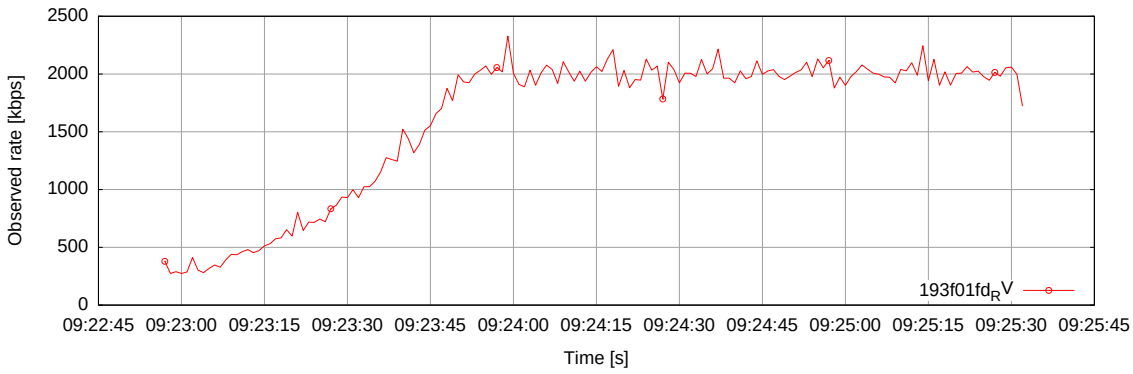


Figure 20: Video stream bandwidth using V4L2Loopback fake YUV file.

Figure 20 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 20 and 18, 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.

5.3 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 [38]. We have modified the source in order to have a hardcoded password making it easier for our needs.

[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.

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.

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.

5.3.1 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 [39].

Dummynet is an standard tool for some Linux distributions and OSX [39]. 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.

5.4 Application Server

Our application server runs the Node.js 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 with JavaScript and uses WebSocket protocol to handle the signaling messages from peer to peer.

5.5 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.

[7] VirtualBox is an x86 virtualization software package.