Real-Time Collaboration on the World Wide Web in a Peer-to-Peer Mode

Emanuel Sérgio Ruivo Almirante

Thesis to obtain the Master of Science Degree in Telecommunications and Informatics Engineering

Supervisor: Prof. Rui António dos Santos Cruz

Examination Committee

Chairperson: Prof. Paulo Jorge Pires Ferreira
Supervisor: Prof. Rui António dos Santos Cruz
Member of the Committee: Prof. Ricardo Jorge Feliciano Lopes Pereira

June 2017
Acknowledgments

Firstly, I want to start by giving my greatest appreciation to my supervisor, Professor Rui Santos Cruz, for all the patience he had and all the help he gave me.

I also want to thank my friends André Pires, André Bernardino, Bruno Barros, Carlos Ribeiro and Gonçalo Borges for all the support and friendship since the beginning of my journey in Instituto Superior Técnico. To my friends Andreia Tibério, Diana Pereira, Francisco Chaves, and Sandra Lopes I want to extend a special thanks for helping and motivating me for all these years. I also thank to the other people that helped me along the way and that left a mark in my life. For that I am deeply grateful for it helped me become what I am today.

Lastly, I wish to thank profoundly to all my family, especially to my grandparents, and to my parents, Manuela Ruivo and Sérgio Almirante, for the support during the good and the bad moments of my life. Without them none of this would have been possible.

Thank you all.
Abstract

The Internet was created with the intention of having a network of computers that could communicate, share and collaborate with each other, no matter where they were physically. Since then, the Internet has evolved exponentially, both in the number of users and in the number of services and activities it offers. Streaming services were one of the services that had an exponentially growth alongside the Internet, being responsible for the majority of the traffic on the World Wide Web (WWW) nowadays. This means that it is necessary to have enough web servers and bandwidth to provide the users with a good enough Quality of Service (QoS), something that may become a challenge when the number of users starts to increase rapidly. As a response to this problem, this thesis presents a collaborative streaming solution using Web Real-Time Communication (WebRTC) that allows content providers to expend less resources on web servers and bandwidth by transforming every user into a serving peer. This solution also permits self-scalability and simplifies the use of Peer-to-Peer (P2P) to all users by requiring only a simple web browser. The developed prototype allows peers using just their web browser, to collaborate with original content and to share content that they had obtained from other servers or peers, with peers that request that content, in a P2P mode. In order to test the required functionalities, the prototype was tested in various scenarios, allowing to conclude that it has the potential to be a viable solution.

Keywords

Peer-to-Peer (P2P), World Wide Web (WWW), Web Real-Time Communication (WebRTC), Real-Time Communications (RTC), Real-Time Collaboration.
Resumo

A Internet foi criada com a intenção de haver uma rede de computadores que pudessem comunicar, partilhar e colaborar entre si, independentemente de onde estivessem fisicamente. Desde então, a Internet tem evoluído exponencialmente, tanto no número de utilizadores, como no número de serviços e actividades que oferece. Os serviços de streaming foram dos serviços que tiveram um crescimento exponencial, em conjunto com a Internet, sendo responsáveis pela maioria do tráfego da World Wide Web (WWW) hoje em dia. Isto significa que é necessário ter servidores web e largura de banda suficientes para providenciar aos utilizadores uma boa qualidade de serviço, algo que pode tornar-se complicado quando o número de utilizadores começa a aumentar rapidamente. Como resposta a este problema, apresenta-se uma solução colaborativa de streaming usando Web Real-Time Communication (WebRTC) que permite aos provedores de conteúdo despender menos recursos em servidores web e largura de banda, transformando todos os utilizadores em peers. Esta solução também permite auto-escalabilidade e simplifica o uso de Peer-to-Peer (P2P) para todos os utilizadores, sendo apenas necessário um simples navegador web. O protótipo desenvolvido permite aos peers colaborarem com conteúdo original ou partilhar conteúdo obtidos do servidor, ou de outros peers, com peers que solicitem esse conteúdo, mas em modo P2P. Por forma a avaliar as funcionalidades requeridas, o protótipo foi testado em vários cenários, concluindo-se que tem potencial para ser uma solução viável.

Palavras Chave

Peer-to-Peer (P2P), World Wide Web (WWW), Web Real-Time Communication (WebRTC), Real-Time Communications (RTC), Colaboração em Tempo Real.
Contents

1 Introduction  
   1.1 Objectives ......................................................... 4  
   1.2 Document Outline ................................................. 5  

2 Fundamentals  
   2.1 P2P Networks and Real-Time Collaboration ......................... 9  
   2.2 WebRTC and RTCWeb ............................................... 10  
      2.2.1 Web Real-Time Communication (WebRTC) ..................... 11  
      2.2.2 Real-Time Communication in Web Browsers (RTCWeb) ........ 12  
   2.3 Signaling in WebRTC .............................................. 15  
      2.3.1 WebSocket .................................................. 15  
      2.3.2 Session Initiation Protocol (SIP) .......................... 16  
      2.3.3 Extensible Messaging and Presence Protocol (XMPP) ........ 17  
      2.3.4 Signaling-On-the-Fly ....................................... 18  
   2.4 Peer-to-Peer Streaming Protocol .................................. 18  
      2.4.1 The PPSP Peer Protocol .................................... 19  
      2.4.2 The PPSP Tracker Protocol .................................. 20  

3 Related Work  
   3.1 Integration of WebRTC and SIP ................................... 23  
   3.2 WebRTC Enabled Multimedia Conferencing and Collaboration Solution ............................................... 24  
   3.3 WebRTC Technology Overview and Signaling Solution Design and Implementation ................................. 27  
   3.4 P2P Live Video Streaming in WebRTC .............................. 29  

4 Architecture of the Prototype  
   4.1 Requirements .................................................... 35  
   4.2 System Design .................................................... 36  
   4.3 System Components .............................................. 39  
      4.3.1 Server Components .......................................... 39  
      4.3.2 Peer Components ........................................... 42
List of Figures

2.1 Types of networks .................................................. 9
2.2 WebRTC and RTCWeb architecture ................................. 11
2.3 WebRTC API with signaling ....................................... 12
2.4 Web browser RTC models of deployment ........................... 13

3.1 Overall architecture of WebRTC with SIP ......................... 23
3.2 WebRTC Enabled Multimedia Conferencing and Collaboration solution .................. 25
3.3 Overall architecture of WebRTC videoconferencing ................. 27
3.4 Message sequence diagram to establish a session .................. 29
3.5 Overall architecture of P2P Live Video in WebRTC ................. 30

4.1 System’s architecture for the proposed solution ................. 36
4.2 System’s participants interaction .................................. 38
4.3 Peer collaborates with content interaction ......................... 39
4.4 Serving Modules: Peer Coordinator, Web Server and Media Server ...... 40
4.5 Peer modules ....................................................... 42

5.1 Interface of the main page of the web application ................. 56
5.2 Interface of the web application when streaming .................. 57
5.3 Interface of the web application when watching a stream .......... 57

6.1 Diagram of Test Scenario 1 ....................................... 62
6.2 Diagram of Test Scenario 2 ....................................... 63
6.3 Diagram of Test Scenario 3 ....................................... 63
6.4 Diagram of Test Scenario 4 ....................................... 64
6.5 RAM used by Firefox in an idle state ......................... 65
6.6 CPU used by Firefox in an idle state ............................. 66
6.7 RAM used by Firefox in scenario 1 .............................. 66
6.8 CPU used by Firefox in scenario 1 ........................................... 67
6.9 RAM used by Firefox in scenario 2 ........................................... 68
6.10 CPU used by Firefox in scenario 2 ........................................... 69
6.11 RAM used by Firefox in scenario 3 ........................................... 70
6.12 CPU used by Firefox in scenario 3 ........................................... 71
6.13 RAM used by Firefox in scenario 4 ........................................... 72
6.14 CPU used by Firefox in scenario 4 ........................................... 73
List of Tables

4.1 Table of association of streams and their sources .................................. 40
5.1 Functionalities added to the Peer Coordinator ........................................ 51
6.1 Latency in scenario 1 .................................................................................. 67
6.2 FPS in scenario 1 ....................................................................................... 68
6.3 Latency in scenario 2 .................................................................................. 69
6.4 FPS in scenario 2 ....................................................................................... 70
6.5 Latency in scenario 3 .................................................................................. 71
6.6 FPS in scenario 3 ....................................................................................... 72
6.7 Latency in scenario 4 .................................................................................. 73
6.8 FPS in scenario 4 ....................................................................................... 74
Listings

5.1 Code for the server Hypertext Transfer Protocol (HTTP) to serve all the files in the public directory ................................................................. 49
5.2 Code for the web server ........................................................................... 49
5.3 Code for the coordinator with a reserved path ......................................... 50
5.4 Code for listing all peers with a certain content ........................................ 52
5.5 Code for listing all peers that are secondary streamers ............................. 52
5.6 Code for registering the peer with the Peer Coordinator ........................... 53
5.7 Video selector attribute .......................................................................... 53
5.8 Example of video in local storage ............................................................. 54
A.1 Code to get the list of peers with the video .............................................. 85
A.2 Code for connecting and requesting data ................................................ 85
A.3 Code for answering calls ...................................................................... 86
A.4 Code for the peer to start streaming ...................................................... 87
A.5 Code for the peer to stop streaming ....................................................... 87
A.6 Code for connecting to a stream ............................................................ 88
A.7 Code to stop watching a stream ............................................................. 88
A.8 Code to see the peers that are streaming ............................................... 89
Acronyms

API Application Programming Interface
BOSH Bidirectional-Streams Over Synchronous HTTP
CDN Content Delivery Network
CERN Conseil Européen pour la Recherche Nucléaire
CPU Central Processing Unit
CSS Cascading Style Sheet
DHT Distributed Hash Table
DOM Document Object Model
DTLS Datagram Transport Layer Security
FPS Frames Per Second
GUI Graphical User Interface
HD High-Definition
HTML5 HyperText Markup Language version 5
HTML HyperText Markup Language
HTTPS Hypertext Transfer Protocol Secure
HTTP Hypertext Transfer Protocol
ICE Interactive Connectivity Establishment
ID Identifier
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IETF</td>
<td>Internet Engineering Task Force</td>
</tr>
<tr>
<td>IPv4</td>
<td>Internet Protocol version 4</td>
</tr>
<tr>
<td>IPv6</td>
<td>Internet Protocol version 6</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IQ</td>
<td>Info/Query</td>
</tr>
<tr>
<td>LEDBAT</td>
<td>Low Extra Delay Background Transport</td>
</tr>
<tr>
<td>MEGACO</td>
<td>Media Gateway Control Protocol</td>
</tr>
<tr>
<td>MMUSIC</td>
<td>Multiparty Multimedia Session Control</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Translation</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>P2P</td>
<td>Peer-to-Peer</td>
</tr>
<tr>
<td>PDF</td>
<td>Portable Document Format</td>
</tr>
<tr>
<td>PNG</td>
<td>Portable Network Graphics</td>
</tr>
<tr>
<td>PPSPP</td>
<td>Peer-to-Peer Streaming Peer Protocol</td>
</tr>
<tr>
<td>PPSP</td>
<td>Peer-to-Peer Streaming Protocol</td>
</tr>
<tr>
<td>PPSTP</td>
<td>Peer-to-Peer Streaming Tracker Protocol</td>
</tr>
<tr>
<td>PTSN</td>
<td>Public Switched Telephone Network</td>
</tr>
<tr>
<td>QoE</td>
<td>Quality of Experience</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>RGB</td>
<td>Red, Green and Blue</td>
</tr>
<tr>
<td>RTCWeb</td>
<td>Real-Time Communication in Web Browsers</td>
</tr>
<tr>
<td>RTC</td>
<td>Real-Time Communications</td>
</tr>
<tr>
<td>RTP</td>
<td>Real-Time Transport Protocol</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>RTSP</td>
<td>Real-Time Streaming Protocol</td>
</tr>
<tr>
<td>SCTP</td>
<td>Stream Control Transmission Protocol</td>
</tr>
<tr>
<td>SDES</td>
<td>Session Description Protocol Security Descriptions</td>
</tr>
<tr>
<td>SDP</td>
<td>Session Description Protocol</td>
</tr>
<tr>
<td>SIP</td>
<td>Session Initiation Protocol</td>
</tr>
<tr>
<td>SMS</td>
<td>Short Message Service</td>
</tr>
<tr>
<td>SRTP</td>
<td>Secure Real-Time Transport Protocol</td>
</tr>
<tr>
<td>STUN</td>
<td>Session Traversal Utilities for Network Address Translation</td>
</tr>
<tr>
<td>TCP</td>
<td>Transport Control Protocol</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>TURN</td>
<td>Traversal Using Relays around Network Address Translation</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>VoIP</td>
<td>Voice over Internet Protocol</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Private Network</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
<tr>
<td>WebRTC</td>
<td>Web Real-Time Communication</td>
</tr>
<tr>
<td>WSS</td>
<td>WebSocket Secure</td>
</tr>
<tr>
<td>WWW</td>
<td>World Wide Web</td>
</tr>
<tr>
<td>XML-RPC</td>
<td>Extensible Markup Language-Remote Procedure Call</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XMPP</td>
<td>Extensible Messaging and Presence Protocol</td>
</tr>
<tr>
<td>YUV</td>
<td>Luminance, Hue and Chrominance</td>
</tr>
</tbody>
</table>
1

Introduction

Contents

1.1 Objectives ......................................................... 4
1.2 Document Outline .................................................. 5
The Internet was created with the intention of having a network of computers that could communicate, share and collaborate with each other, no matter where they were physically. Its focus, in the beginning, was to perform simple tasks such as, creating and sharing documents or sending email messages [1]. The success of this simple infrastructure paved the way for one of the biggest changes in human history.

This change would reach its most important milestone with the proposal of the World Wide Web (WWW), originated at the Conseil Européen pour la Recherche Nucléaire (CERN), by Tim Berners-Lee [2]. The Internet was revolutionized, and evolved from a simple infrastructure, where only the creation and sharing of simple documents and sending email messages was possible, to something that, additionally, would allow access to websites, send text messages, transfer binary files (programs/applications, images, audio, video), and much more.

The evolution of the Internet and of the WWW has not stopped since, and nowadays it is possible to pursue a multitude of activities, such as online shopping, audio and video calls, play video-games online, live-streaming, on-demand multimedia, and even the virtualization of services (e.g., software, application platforms or infrastructures) are possible as well. Almost every service we can think of is now accessible online, which shows how important the WWW has become to society.

As aforementioned, one of the most important aspects of the Internet is the possibility to collaborate with anyone in the network. With the introduction of the WWW and the expansion of the Web, millions of people from around the World can engage in real-time collaboration with each other, using technology present in applications such as Google Docs, Overleaf or Slack. Real-time collaboration changed how professionals from various fields would work with each other, allowing for a more efficient collaboration by getting an answer immediately. However, most of these applications use the classic client-server paradigm which means that there is a single point of failure, the server may become a bottleneck, among other limitations, causing problems of scalability and, sometimes, of Quality of Service (QoS).

A possible solution to mitigate this problem is by using Peer-to-Peer (P2P). Theoretically, a P2P system works better as more users are online cooperating. This would mean that the QoS would stabilize no matter how many people used the application, because a user would act simultaneously as a client and as a server, reducing, or even eliminating, the workload from a main server. Although P2P systems are used by millions of users nowadays, they have some disadvantages, like, for example, needing extra software or plugins to be installed, which may cause compatibility problems between different devices.

But recently a new technology named Web Real-Time Communication (WebRTC) has emerged. This technology allows web browsers to connect directly in a P2P fashion, providing real-time device-to-device communication without the need to install any software, besides the web browser, or plugins. Although not yet completely standardized, at time of writing, WebRTC basic functions have already been adopted by the majority of web browsers, such as Google Chrome, Mozilla Firefox, and Opera, immediately delivering WebRTC to a huge user base, and making it attractive to developers.
Considering that video streaming is one of the most bandwidth consuming activities, and one where it is necessary to spend more money in servers, bandwidth, Content Delivery Network (CDN), among others, to provide a good enough service for all the users, it would have a great advantage from the implementation of this technology. Due to this fact, live or on-demand video streaming would greatly benefit from this technology, and would be a good way to test if a system based on it would still provide a good enough service, while reducing costs.

Another important factor is the possibility of each user to collaborate with its own produced video, live or prerecorded, in real-time, or by mixing a stream of another peer with its own stream. All of this happens without having to upload anything to a central server or having to wait to be able to stream the video, such as it happens nowadays with “torrent”-type\(^1\) services, for example. By using WebRTC, there can be an almost complete reduction on the server bandwidth and on the time waited to stream a video, by eliminating the process of creating a “torrent”, for example, making it easy to collaborate with other users on the network.

Using both P2P mode, to provide real-time collaboration and WebRTC in a browser, to simplify the use of P2P for everyone, and enable it to be used in any device, at any time, this system would present itself as a self-scalable solution by decreasing the high costs with bandwidth, because each user would also act as a server sharing its own resources. Additionally, the combined use of these technologies provides the necessary tools to develop applications that are easy to integrate with the ones already being used, making it simple to transition to a “universal” P2P system.

1.1 Objectives

The primary objective of this project is to develop a prototype of a system that allows real-time P2P collaboration by taking advantage of the characteristics of WebRTC. With WebRTC, real-time P2P collaboration will be possible using only a simple web browser. This will turn the users into content providers too, causing them to also spread the content they are accessing.

Just referring to a system that provides a real-time P2P collaboration is very vague and can lead to some confusion. The solution proposed in this work will be considered a real-time P2P collaboration using live video streams from peers, i.e., where every peer can transmit its original stream, as long as it is authorized. A peer can also modify the stream that receives from other peers and transmit that “combined” stream, as if it was its own original stream, to other peers.

The solution for the prototype is composed of three modules in a single node, for the following functionalities:

(a) a “Peer Coordinator” entity, that manages the registrations of peers, their authorizations to collab-

---

\(^1\) BitTorrent-based P2P [3] services for file sharing, file synchronization, or even media streaming.
orate with video and manages the network of peers; (b) a normal Web Server, that the peers will access, providing the website and the web application, and (c) a Media Server, that will act as a super-peer, i.e., it will always have published videos available for the peers and will always be online, so that there is always at least one source of contents in the P2P network.

Finally, any user will interact with the system through a web application loaded in the browser, which will provide an interface through which the user watches the video and also contributes with own content streams.

1.2 Document Outline

This document is structured as follows:

- **Chapter 2**: presents the key related technologies;
- **Chapter 3**: presents some related works already done in this field;
- **Chapter 4**: describes the requirements and the architecture of the proposed solution;
- **Chapter 5**: describes how the solution was implemented;
- **Chapter 6**: describes the tests scenarios, the evaluation of the solution and discusses the results;
- **Chapter 7**: draws conclusions about the work developed in this thesis, lists the system limitations and points out areas of development for future work.
2 Fundamentals

Contents

2.1 P2P Networks and Real-Time Collaboration ........................................ 9
2.2 WebRTC and RTCWeb ................................................................. 10
2.3 Signaling in WebRTC ................................................................. 15
2.4 Peer-to-Peer Streaming Protocol .................................................... 18
This chapter provides some insight about the technologies that will be used in the project. It starts by giving an overview of the basics behind a P2P network and how real-time collaboration works using these networks. It is also explained how the WebRTC and the Real-Time Communication in Web Browsers (RTCWeb) technologies can be used to provide P2P capabilities to a browser, as well as some of the signaling technologies compatible with them. Finally, the Peer-to-Peer Streaming Protocol (PPSP), a new protocol that standardizes the communications in a P2P network, is described.

2.1 P2P Networks and Real-Time Collaboration

Peer-to-Peer (P2P) is an alternative model to the traditional client-server paradigm (see Figure 2.1). It consists in a group of nodes that cooperate on the Internet in a way to reach a common goal, e.g., exchange contents. But P2P can also be used to establish live multimedia communication, e.g., videoconferencing, or share resources based on P2P concepts [4].

The P2P network works on top of the physical network of the Internet, more specifically it is overlayed on the Internet Protocol (IP) networks. The P2P overlay network makes a “direct connection” between two random users that may or may not know each other. This “direct connection” is established as if the two users are directly connected to each other without routers or servers or other nodes in-between. It is an abstraction of the physical network, to simplify [5].

Networks based on P2P distinguish themselves by making each peer a client and a server simultaneously. In this model the peers send requests and answer to incoming requests from other peers, instead of only sending requests and waiting for an answer [6].

The implementation of a P2P network can be done with one or more central entities or without them. The first mode is commonly known by hybrid P2P and the second by pure P2P [7].

![Figure 2.1: Types of networks, adapted from [7]](image)

A hybrid P2P network, has a central entity, better known as a “Tracker”. This tracker has a list of the peers on the network and what content each of them is sharing, among other informations about the
peers, the content, and the network. When a peer wants to download something, it sends a request to the tracker, and the tracker helps the peer to search for other peers that have the desired content.

In contrast, a pure P2P network does not have any central server (tracker), it only has peers. Hence, each peer maintains a list of their neighboring pairs, knows the content that each one has, and it connects to them without help of any tracker.

Concerning real-time collaboration in a P2P system, the peers connect directly to each other, obviously, and each peer transmits to the other peers the content they want to send, such as text, audio, video, binary files, and others. This will only depend on the web application and what type of collaboration is allowed to happen between the peers. Some applications will be of videoconferencing only, others will permit a whiteboard-type collaboration, and others may group all of these features.

In the event of an application allowing the transmission of live video streaming there will be an original content provider or source peer that will stream the live video to other peers, but the original peer does not transmit the content to all peers. Instead, the original peer only transmits to a limited number of peers, for example, five, and then those peers will also transmit to a limited number of other peers, and so on. In this situation, there might be some delay, but nothing that will relevantly affect the user’s experience [8].

Nowadays, the most famous P2P network is the BitTorrent, dedicated to the distribution of files over the Internet. In 2005 it accounted for 35% of all Internet traffic and more than 50% of all P2P traffic [3]. BitTorrent is the proof that P2P networks can be an alternative to the traditional server-client networks.

2.2 WebRTC and RTCWeb

Web Real-Time Communication (WebRTC) [9] and Real-Time Communication in Web Browsers (RTCWeb) [10] are two different technologies that complement each other, in order to achieve a common goal: to support direct browser-to-browser communication.

WebRTC is an Application Programming Interface (API) that provides programmers with the tools to build applications that communicate directly, creating a browser-to-browser P2P connection which can be used for a multitude of things, like videoconferencing, stream video (live or on demand), exchange files, among others [11].

RTCWeb is a protocol that encompasses various existing protocols, to make possible that the WebRTC applications communicate between themselves with audio, video, and data in a P2P mode.

Together, WebRTC and RTCWeb provide an environment where JavaScript embedded in any website (that is viewed in a compatible web browser and authorized by its user) is able to set up communication in a P2P fashion, using audio, video and data [10].

Figure 2.2 illustrates the overall architecture of WebRTC and RTCWeb.
2.2.1 WebRTC

Basically, WebRTC is an API that puts real-time communications capabilities into web browsers. These capabilities will then be available to developers through the combination of HyperText Markup Language version 5 (HTML5) tags and JavaScript APIs.

Because the WebRTC API includes HTML5 and JavaScript, it is possible to have web applications with functionalities similar to Skype but without the need to install anything else besides using a common web browser [11].

The components to implement real-time communications capabilities of the WebRTC API are accessed with JavaScript APIs. The most important API, and the ones necessary to establish a live video streaming, are:

- **MediaStream**: allows the web browser to access the camera and microphone;
- **PeerConnection**: establishes peer-to-peer communications, allowing two peers to communicate directly;
- **DataChannel**: sets up a bidirectional channel through which data can be exchanged between two peers.

These three API will allow two or more users to communicate browser-to-browser, in a real-time and P2P mode [12]. To start a WebRTC session it is necessary to complete four main steps:

1. Web browser client obtains local media;
2. Sets up connection between the web browser and the other peer through signaling;
3. Attach the media and data channels to the connection;

4. Exchange the session description from each other.

After the above steps performed, as illustrated in Figure 2.3, the media stream will start being exchanged through the real-time P2P media channel. All of these steps are implemented by the WebRTC API.

![Figure 2.3: WebRTC API with signaling, adapted from [13]](image)

The signaling part in Figure 2.3 is also an important component of the system. WebRTC needs a signaling mechanism to coordinate communications and to send control messages. Because signaling methods and protocols are not specified by the WebRTC API, developers can choose the messaging protocol they prefer, as long as it is a two-way communication channel. Signaling serves to exchange three types of information [13]:

- **Session control messages**: initialize or close communication and report errors;
- **Network configuration**: let the outside world know the computer’s IP address and port;
- **Media capabilities**: decide the codecs and resolutions that can be handled by the web browser and the web browser it is trying to communicate with.

The signaling must be successful in order for the P2P connection to begin.

The WebRTC API is, at time of writing, being standardized by the World Wide Web Consortium (W3C) and it is already supported by the main web browsers (Chrome, Firefox, Opera) and platforms (Android, iOS) [9].

### 2.2.2 RTCWeb

RTCWeb is a protocol specification, standardized by the Internet Engineering Task Force (IETF), that defines a method of exchanging data over the Internet [10].
This protocol specifies a set of protocols to allow an implementation to communicate with another implementation using audio, video and other data. The participants will communicate in a P2P fashion, meaning they will be directly connected, without intermediaries. The models of deployment for which the protocol is best suited are the triangle and the trapezoid [11].

![Diagram of Web browser RTC models of deployment](image)

**Figure 2.4:** Web browser RTC models of deployment, adapted from [11]

As it can be observed in Figure 2.4, the media path goes directly between the web browsers, so it is critical to be in accordance with the RTCWeb protocol. The signaling path is not as critical because it goes through servers that can modify or translate the messages as necessary. It is noteworthy that the signaling path in the trapezoid model is not of exclusive use of the servers, and the web browsers can communicate through it, in the beginning of a session, to establish the Media Path, for example.

In the triangle model, the signaling runs over Hypertext Transfer Protocol (HTTP) or WebSocket. In the trapezoid model the servers need to agree on the signaling mechanism. Session Initiation Protocol (SIP) or Extensible Messaging and Presence Protocol (XMPP) are two examples of protocols that can be used [11].

Between the web browser and the server a standards-base or proprietary protocol can be used. For example, if both servers use SIP as signaling mechanism then SIP over WebSocket can be used to communicate between the application running in the web browser and the web server [10].

As defined in [14], it is necessary that both User Datagram Protocol (UDP) and Transport Control Protocol (TCP) with the ability to use Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6) are available to the implementations.

Also necessary is the support of a full Interactive Connectivity Establishment (ICE) implementation in order for both participants to be able to communicate when they are behind a Network Address Translation (NAT). For this, Session Traversal Utilities for Network Address Translation (STUN) or Traversal Using Relays around Network Address Translation (TURN) servers will be used, with the preference being TURN servers. If the participants are behind a firewall that blocks all UDP traffic, the mode of TURN that uses TCP between the client and the server must be supported, and the mode of TURN that uses Transport Layer Security (TLS) over TCP between the client and the server must also be supported.
The transport of media is typically done by secure Real-Time Transport Protocol (RTP), while key exchange must be done using Datagram Transport Layer Security-Secure Real-Time Transport Protocol (DTLS-SRTP). For data transport over the data channel, Stream Control Transmission Protocol (SCTP) over Datagram Transport Layer Security (DTLS) over ICE must be supported. The negotiation of this transport is done in Session Description Protocol (SDP), so the multiplexing of DTLS and RTP over the same port pair must also be supported [14].

One of the most fundamental aspects is the connection management. In this aspect of the RTCWeb protocol it is important to have interoperability and freedom to innovate. It is also essential that the methods, mechanisms and requirements needed to set up, negotiate and tear down connections follow these principles:

- The media negotiations will be capable of representing the same SDP offer/answer semantics that are used in RFC3264 [15], in such a way that it is possible to build a signaling gateway between SIP and the media negotiation;

- It will be possible to gateway between legacy SIP devices that support ICE and appropriate RTP/SDP mechanisms, codecs and security mechanisms without using a media gateway. A signaling gateway to convert between the signaling on the web side to the SIP signaling may be needed;

- When a new codec is specified, and the SDP for the new codec is specified in the Multiparty Multimedia Session Control (MMUSIC) working group, no other standardization should be required for it to be possible to use that in the web browsers. Adding new codecs which might have new SDP parameters should not change the APIs between the web browser and JavaScript application. As soon as the web browsers support the new codecs, old applications, written before the codecs were specified, should automatically be able to use the new codecs where appropriate with no changes to the JavaScript applications.

These aspects of the RTCWeb protocol described are the most important. There are other secondary aspects, like the audio and video codecs or the security of the communications, that should be considered, but are not as important as the ones mentioned in [10].

To conclude, the RTCWeb protocol is always used in collaboration with the WebRTC API to be implemented, with the WebRTC API being merely a facilitator to deploy the requirements of the protocol, and it may function in any type of device, as long as the requirements are fulfilled.
2.3 Signaling in WebRTC

Even though WebRTC has the objective of connecting users directly without any servers as intermediaries, to establish a connection between two or more users it is still necessary to have servers so that users can exchange metadata to coordinate communication. This is called signaling.

Signaling in WebRTC has the job of passing messages back and forth between clients, that exchange information such as:

- **Session control messages**: to initialize or close communication and report errors;
- **Network configuration**: to let the outside world know the computer’s IP address and port;
- **Media capabilities**: to decide the codecs and resolutions that can be handled by the web browser and which browser it is trying to communicate with.

WebRTC does not implement any type of signaling, leaving this aspect to the will of the developers. This happens because different applications may need or prefer to use different signaling protocols, like SIP [16], XMPP [17], WebSocket [18], or even a custom one.

2.3.1 WebSocket

Web applications that needed bidirectional communication between a client and a server were required to abuse the HTTP to poll the server for updates while sending upstream notifications as distinct HTTP calls. This form of bidirectional communication brought a variety of problems, including:

- The wire protocol has a high overhead, with each client-to-server message having an HTTP header;
- The server is forced to use a number of different underlying TCP connections for each client: one for sending information to the client and a new one for each incoming messages;
- The client side script is forced to maintain a mapping from the outgoing connections to the incoming connection to track replies.

The WebSocket protocol [18] offers a simpler solution by using a single TCP connection for bidirectional traffic. This protocol, combined with the WebSocket API [19], can provide an alternative to HTTP polling for bidirectional communication from a website to a remote server.

The protocol operation consists of an opening handshake between the client and the server. If the handshake is successful then the data transfer can begin. The data transfer occurs through a bidirectional channel, and each side can send data at will, without any dependence on the other. In the
end of the communication a closing handshake is sent and the connection is closed, with all the data transfer being stopped.

This protocol is designed to substitute the existing bidirectional communications that use HTTP as a transport layer. These types of communications were implemented as trade-offs between efficiency and reliability because HTTP was not meant to be used for bidirectional communications. The WebSocket protocol works over HTTP ports 80 and 443 and supports HTTP proxies and intermediaries.

The design of the WebSocket protocol does not limit it to HTTP, meaning that future implementations may use a simpler handshake over a dedicated protocol. Being this flexible is an important part of the protocol because the traffic patterns of interactive messaging do not closely match standard HTTP traffic which can induce unusual loads on some components.

Summing up, the WebSocket protocol provides a mechanism for web applications that need bidirectional communication with servers that do not support the opening of various HTTP connections. It can be used for games, stock tickers, multiuser applications with simultaneous editing, user interfaces exposing server-side services in real-time, among others.

2.3.2 Session Initiation Protocol (SIP)

Session Initiation Protocol (SIP) [16] is a protocol used for the establishment of communication sessions between two or more users. SIP creates, modifies, and terminates these communication sessions. These sessions can be Internet telephone calls, multimedia distribution, and multimedia conferences. The protocol is text-based, modeled on the request/response paradigm used in the HTTP.

There are five characteristics of establishing and terminating multimedia communications that SIP supports:

- **User location**: determination of the end system to be used for communication;
- **User availability**: determination of the willingness of the called party to engage in communications;
- **User capabilities**: determination of the media and media parameters to be used;
- **Session setup - “ringing”**: establishment of session parameters at both called and calling party;
- **Session management**: including transfer and termination of sessions, modifying session parameters, and invoking services.

SIP can also be used with other IETF protocols in order to have a more complete multimedia architecture and provide a better experience to the users.

Usually, in these systems, SIP is used together with RTP, for transport of real-time data and to provide QoS feedback, the Real-Time Streaming Protocol (RTSP), for controlling the delivery of streaming
media, the Media Gateway Control Protocol (MEGACO), for controlling gateways to the Public Switched Telephone Network (PSTN), and the SDP, for describing multimedia sessions.

Although SIP should be used together with other protocols, to provide a complete service to the users, its basic features and operation are not dependent of any other protocol.

Concerning the use of SIP as a signaling protocol for WebRTC, it is used over WebSocket [20]. WebSocket works as a transport protocol carrying the SIP messages, instead of the proprietary messages. This is necessary because the main transport protocols of SIP, i.e., UDP, TCP, TLS, and SCTP, are not directly accessible from the web browser. This means that there is no possibility to have a two-way real-time communication between clients and servers in web-based applications without transporting the SIP messages over WebSocket.

Also, SIP over WebSocket enables web communications with SIP networks, mobile and fixed phones, by giving web browsers the same capabilities as a mobile phone, like audio, video and Short Message Service (SMS).

2.3.3 Extensible Messaging and Presence Protocol (XMPP)

XMPP [17] is a communications protocol that allows real-time exchange of data between two or more network entities. It is based on Extensible Markup Language (XML), which makes it attractive for applications that need structured messages and rich hypermedia.

It started by being used only for instant messaging, presence information and contact list maintenance by the Jabber community [21]. But due to its extensibility, the protocol has also been used for Voice over Internet Protocol (VoIP), file transfer, video, and others, by the addition of extensions [22,23].

The bidirectional communication of XMPP is done over HTTP, using the Bidirectional-Streams Over Synchronous HTTP (BOSH) protocol, that defines how arbitrary XML elements can be transported efficiently and reliably over HTTP in both directions between a client and server [24]. This communication can also be attained by WebSocket [25].

Each node on a XMPP network uses XMPP content namespaces. For client-to-server communication it is used jabber:client. For server-to-server communication it is used jabber:server. It is used jabber instead of, for example, xmpp, because XMPP was created by the Jabber community, as aforementioned.

When using XMPP, users are identified by an Identifier (ID), that is formed using a combination of the username and the server name to form a username@server type of ID. This allows to easily exchange messages with users in different servers.

The users profiles are represented using the protocol Data Forms [26] and are registered and updated through Info/Query (IQ) requests to the server. The server will answer with an IQ response that has information in order to retrieve the registration fields needed to complete the users profile.
Concerning the connection between the user and the XMPP server, this can be performed using the same web server the user was already connected to. This will put a lot of stress on the web server, making it responsible for handling a lot of connections, and greatly impact the system performance. When using web applications, users can connect directly to the XMPP servers using Strophe.js, a JavaScript library [27].

2.3.4 Signaling-On-the-Fly

A very interesting concept to solve the problem of signaling is Signaling-On-the-Fly, also known as “SigOfly” [28]. It uses the ID of each peer and JavaScript, with the objective of achieving interoperability between any WebRTC service provider domains and providing inter-domain communications.

With “SigOfly”, WebRTC applications use JavaScript to implement signaling protocol stacks. The signaling protocol stack can be selected, loaded and instantiated during runtime. This characteristic is the one that enables signaling protocols to be selected by each WebRTC conversation, to ensure full signaling interoperability among peers. Basically, the peers download all the code necessary to communicate with each other.

The biggest advantage of “SigOfly” is the fact that users are not restricted to only one type of signaling implementation. It also allows for multiparty conversations with more than one user coming from different domains, through a Mesh Topology with a Hosting peer or a Multipoint Control Unit based Topology with a Hosting peer.

2.4 Peer-to-Peer Streaming Protocol

As the P2P streaming traffic continues to experience a substantial growth, more providers are transitioning to P2P systems. Some CDN providers have started to use P2P systems and technologies to distribute their streaming content [29].

Most of those systems use proprietary P2P protocols, which means that new developers need to develop their own protocols, and the lock-in effects lead to substantial integration difficulties with other players, e.g., CDN. This means that there was a need for an open and standard streaming protocol for P2P networks.

To suppress that necessity, the PPSP [30] was developed. It standardizes the communication operations in every P2P streaming systems to solve the aforementioned problems. This protocol includes the Peer-to-Peer Streaming Peer Protocol (PPSPP) [31] and the Peer-to-Peer Streaming Tracker Protocol (PPSTP) [32] protocols, which will be described next.
2.4.1 The PPSP Peer Protocol

PPSP is designed to transmit, in a streaming fashion, the same content to a group of peers. The protocol is based on the P2P model where a peer consuming content also acts like a server and disseminates that content to other peers, to create a system where everyone can provide upload bandwidth. PPSP can stream prerecorded content and live audio and video content [31].

The protocol is generic, so it can run directly on top of UDP, TCP, or other protocols. Currently, PPSP runs on top of UDP using Low Extra Delay Background Transport (LEDBAT) for congestion control. LEDBAT makes possible for PPSP to deliver the content without disrupting other tasks that the peer might be doing and that use the peers network connection [33].

To prevent the interruption of the streams by malicious peers, the design of PPSP has a short time-till-playback for the end user. To achieve this, the content in PPSP is identified by a single cryptographic hash that is the root hash in a Merkle hash tree calculated recursively from the content. This tree, that can be used for prerecorded or live content, allows every peer to detect when another peer tries to distribute fake content [34].

One of the big problems with P2P systems is the fact that some peers only download content and never upload to other peers. PPSP has flexible and extensible mechanisms to prevent this and to promote user cooperation. It also offers the possibility to have different schemes for chunk addressing and content integrity protection besides the defaults, so that each case has a scheme fit for it, and the possibility to work with different peer discovery systems, like centralized trackers or fast Distributed Hash Table (DHT).

Relatively to peers, by default, PPSP only maintains a small amount of state per peer. It also assumes that a peer gets a list of peers from a centralized tracker beforehand, and only after having that list the peer is able to connect to other peers, request content, and discover other peers disseminating the same content.

The overall operation of the PPSP can be summarized in three actions:

- **Join a swarm**: to join a swarm, the peer starts by registering with the tracker, following the specification of the PPSTP [32]. Then the tracker will provide the peer with a list of peers in the swarm. After that, the peer will begin to exchange messages with other peers, using the PPSP, so that it can connect to them. The peers also exchange information about the content and availability of it;

- **Exchange chunks**: a peer sends messages requesting chunks of a certain content. Peers that have those chunks will answer this request. The first peer receives the chunks, checks the integrity of them, and when it finishes to receive them, it sends a message to the other peers telling them that the reception of the chunk was successful. A peer is not obliged to answer every request and can decide when it accepts requests for the chunks it has. After the complete download of the
content, a peer becomes a seeder and only sends messages to peers that still do not have the complete content, to only send messages to peers that still need the content;

- **Leave a swarm:** when a peer wants to leave a swarm, it sends a specific message to all its peers and also sends a message to the tracker, following the specification in the PPSTP, to unregister from it. The other peers remove the peer that left from their list. In case a peer crashes unexpectedly, the other peers will remove it from the list as soon as they detect that it no longer sends messages.

### 2.4.2 The PPSP Tracker Protocol

Peers are usually a user device that participates in sharing media content, receiving and transmitting content. They are organized in one or more swarms. In each swarm there are only peers that are streaming the same content at any given time. The role of the tracker is to coordinate the peers in a swarm. Trackers maintain a list of peers that are storing chunks for a specific content, answer queries from peers and collect information about the state and the activity of peers.

To be able to do this, it is necessary to have a form of communication between peers and trackers. Nowadays, this communication is performed using a variety of different protocols, making difficult to achieve interoperability between every device.

It was with this in mind that there was the development of an open protocol, PPSTP [32], that standardizes all the communication process between peers and trackers and allows interoperability. It allows the peers to send meta information to the trackers, report streaming status, and obtain lists with the peers in a swarm. A typical PPSTP session, if a peer wants to receive content, consists in:

- A peer connects to a tracker, registers in that tracker, and joins a swarm;
- That peer receives a list of other peers in the swarm from the tracker;
- Peer starts communicating directly with the other peers without the intervention of the tracker.

If a peer wants to share content, the session consists in:

- A peer connects and registers on a tracker;
- The peer sends information about the swarms it belongs to to the tracker;
- Peer waits for other peers to connect with them.

While a peer is connected to a swarm it needs to periodically report its status to the tracker. This allows the tracker to know that the peer is still active and to update the information regarding that tracker. If a peer is disconnected in any way, it can always communicate with the tracker and rejoin the corresponding swarms, to continue the previous activity [31, 32].
3

Related Work

Contents

3.1 Integration of WebRTC and SIP .......................... 23
3.2 WebRTC Enabled Multimedia Conferencing and Collaboration Solution .......... 24
3.3 WebRTC Technology Overview and Signaling Solution Design and Implementation 27
3.4 P2P Live Video Streaming in WebRTC ........................ 29
This chapter presents some projects that were developed using some of the technologies referred in Chapter 2, and that are relevant for the project developed in this thesis. Most of the works described in this chapter were implemented in a controlled environment, a laboratory environment, so to say, but they can be easily adapted to real world situations. For this, they are considered relevant and an important learning tool for this thesis.

### 3.1 Integration of WebRTC and SIP

The authors in project [35] show how to surpass the difficulties in making interoperable the SIP and WebRTC integration, to achieve RTC sessions between browser-to-browser, browser-to-SIP communicator, or web browser to legacy phone, for example.

WebRTC enables browser-to-browser RTC connection, but it requires a signaling protocol. In this case, SIP was the one used, because it solves the issue of interoperability with the already built SIP systems.

The integration of WebRTC and SIP carries some problems, especially at the media and signaling plane. For the signaling plane, the authors provided the SIP signaling functionalities through SIP WebSocket API. For the media plane, it was used a media gateway, namely the mediaproxy-ng and webrtc2sip.

For the implementation of this solution, it was necessary a WebRTC client, a SIP signaling server and a media gateway. The architecture of the solution is shown in Figure 3.1.

![Overall architecture of WebRTC with SIP](image)

**Figure 3.1:** Overall architecture of WebRTC with SIP, adapted from [35]

The WebRTC client signaling functionalities are implemented using WebSocket SIP API, so that WebRTC-to-WebRTC or WebRTC-to-SIP sessions can be established. The WebRTC client uses a
WebRTC SIP application created by the authors and it has, as expected, WebSocket and SIP characteristics for signaling.

The signaling server used is a Kamailio SIP proxy server, that supports the WebSocket protocol, because a WebRTC client uses the WebSocket protocol as signaling mechanism, so it is necessary to have a SIP entity that has a WebSocket and SIP protocol integrated.

It was also necessary to have a media gateway, to work as a translating mechanism. In this case, a webrtc2sip gateway was used. This is due to the fact that WebRTC demands support of protocols that are not present in simple SIP clients and because different web browsers do not support the same key exchange mechanisms, so it is necessary an intermediary (webrtc2sip) to translate the different protocols supported by Firefox (supports DTLS-SRTP), Chrome version 33.0 (supports Secure Real-Time Transport Protocol-Session Description Protocol Security Descriptions (SRTP-SDES)) and a normal SIP client (supports Zimmermann Real-Time Transport Protocol-Secure Real-Time Transport Protocol (ZRTP-SRTP)).

In the experimental tests, the authors noticed that sessions between peers using the same web browser would work without any problems but sessions between peers using different web browsers needed to access the media gateway for translation, as explained before. Although for the case of peers using different web browsers it was necessary to have an intermediary gateway, and they were not directly connected to each other, it could be considered that the integration of WebRTC and SIP was successful and it was possible to establish communication between two peers.

This implementation shows that it is possible to integrate the WebRTC protocol with other technologies relevant to the building of a communication system compatible with all users, using only free and open-source components and technologies.

3.2 WebRTC Enabled Multimedia Conferencing and Collaboration Solution

In [36] it is presented a solution for interoperation and migration between SIP-based systems and the WebRTC by an implementation and enhancement of an open-source videoconferencing system.

The architecture for the system can be observed in Figure 3.2. This architecture consists of three important components:

- **Media Server**: provides all media handling functionalities such as audio, video and text encoding and decoding, media mixing, web broadcasting and finally, recording and playback of files. It is open source and can be divided into open source encoder libraries, video and audio mixers, FFmpeg and a Extensible Markup Language-Remote Procedure Call (XML-RPC) server;
• **mcuWeb application**: open-source Java application based on JavaServer Pages and HTTP Servlets. It is responsible for commanding the media server through the XML-RPC interface. Also provides an administration web interface for managing and operating the service;

• **SIP Application Server (Focus)**: responsible for the SIP signaling and running the mcuWeb application, where the open source version of the multi-videoconferencing system is using JBoss Application Server with SIP Servlets.

![Figure 3.2: WebRTC Enabled Multimedia Conferencing and Collaboration, adapted from [36]](image)

The SIP Call Server Kamailio, that represents a SIP Proxy Server, SIP Registrar and Location Server, was implemented to enable the registration and authentication of legacy SIP User Agents and for routing SIP messages (requests and responses) between these User Agents and the Application Server.

As soon as Kamailio receives a SIP INVITE message sent from a SIP User Agent to join a video-conference, it forwards the request to the SIP Application Server. The SIP signaling is handled by the mcuWeb application. Finally, the mcuWeb commands the Media Server via the XML-RPC interface to establish the RTP media session with the SIP User Agent.

Kamailio was configured to also support WebSockets, allowing it to act as a WebSocket server capable of sending and receiving SIP over WebSockets messages. Several open-source SIP WebRTC clients were implemented to the system. These clients are written in JavaScript and provided by a locally implemented Apache web server.

In this system the WebSocket signaling procedure is as follows:

• The participant downloads the WebRTC client from the Web server using Google Chrome;

• A TCP connection is established between Kamailio and the WebRTC client to register the WebRTC client on the Kamailio Web browser;
• Next, an opening WebSocket handshake to switch protocols from HTTP to WebSocket, where Kamailio and the WebRTC client define and specify SIP as a WebSocket sub-protocol.

• From now on and during this session, all SIP messages will be transferred over the WebSocket protocol.

Basically, the main function of the WebSockets server, Kamailio, is to translate SIP over WebSockets messages coming from WebRTC clients to SIP messages, and then forwarding them to the Application Server and vice versa.

Besides implementing videoconferences with High-Definition (HD) resolution, this implementation also allows participants to interact and deal with documents within a live videoconferencing session. The slide presentations development can be divided into the following components:

• File upload Servlet: used to upload the document to the videoconferencing server;

• Converters: four converters that can be used to convert PowerPoint, Word, Excel, Portable Document Format (PDF) documents and different types of images to Portable Network Graphics (PNG) images;

• XML-RPC methods: the XML-RPC interface was extended to enable the transfer of new parameters needed to add a document to a videoconferencing session and to change the slide number;

• FFmpeg decoder and converter: to decode the PNG images and convert the Red, Green and Blue (RGB) color space to the Luminance, Hue and Chrominance (YUV) representation used in live video streaming applications;

• File download Servlet: used to download the document from the videoconferencing server;

• HyperText Markup Language (HTML) forms: Several HTML submit buttons and select elements that were programmed in the conference JavaServer Page to allow the interconnection between participants and the videoconferencing system to upload a document, to change the number of the streamed slide, to magnify or shrink a portion of a live streamed document, to download or delete a document from the videoconferencing server;

• Zooming module: allows participants to magnify or shrink a portion of a streamed slide;

• Service logic: the service logic of the mcuWeb application and the media server was extended to enable the developed components to work together and cooperate to provide the slide presentation service.

Regarding security of the communications, the authors of this project encrypt the signaling part of the service using the TLS protocol. The WebSocket Secure (WSS) protocol is used by WebRTC clients
to transfer SIP messages. Moreover, the media flow between the media server and SIP User Agents or SIP WebRTC clients is done using the Secure Real-Time Transport Protocol (SRTP), where the Session Description Protocol Security Descriptions (SDES) is used for SRTP key negotiation and for association management. Lastly, the web management interface can be only accessed using the Hypertext Transfer Protocol Secure (HTTPS).

In conclusion, the project described in [36] implemented, developed and enhanced an open-source solution of a voice and multimedia application that can be used for videoconferencing, live web broadcasting and live slide presentations, making possible for legacy SIP User Agents and modern SIP WebRTC clients to join the same session.

### 3.3 WebRTC Technology Overview and Signaling Solution Design and Implementation

Sredojev et al. [12] implemented a videoconferencing and chat application that was able to connect peers through the web browsers without any external plugins, using WebRTC.

This work was made considering communications inside an enterprise and its branches, where the communications were carried out through a Virtual Private Network (VPN). Even so, the techniques applied in this project can be used in more scenarios than just inside a controlled environment, like the one where this project was implemented.

The overall architecture of the project is very simple, as can be observed in Figure 3.3. It is only necessary a signaling server, two peers and a way for the peers to communicate (signal) with the server.

![Figure 3.3: Overall architecture of WebRTC videoconferencing, as seen in [12]](image)

The Signaling Server is a WebSocket server, written in Node. The decision of implementing the server in Node was due to the fact that this is an open-source, cross-platform runtime environment for the server-side and networking applications, that optimizes the throughput and scalability of an application, something essential for when the server needs to handle more than two peers at the same time. The
server also (a) handles the control messages: “initialize”, “initiator”, “got user media” and “peerChannel”; (b) handles media information messages: “offer” and “answer”; (c) handles network information messages: “candidate”, and (d) manages the peers.

As expected, the application for the peers was implemented using the WebRTC API. This API is used, mainly, for a P2P connection between the web browsers of the peers. Before the connection is established, local and remote descriptions of the audio and video media information are exchanged. For this it is used the `setLocalDescription()` method and the `setRemoteDescription()` method, that are in the API.

For communicating with the Signaling Server, it used the WebSocket protocol, which allows to establish a session between the peer and the server. The WebSocket protocol maintains the session open until it is closed.

The signaling solution of this project used four types of control messages, as mentioned before:

- initialize;
- initiator;
- got user media;
- peerChannel.

Depicted in Figure 3.4 is a diagram of a session establishment between two peers in this solution.

The initialize message is the one sent by any peer that wants to signal the server, to register itself, as it can be seen, in Figure 3.4. In this case, Peer1 is the one to initiate the session, as confirmed by the initiator message sent by the Server. After knowing that it is the initiator of the session, Peer1 will configure the user media following the reception of the got user media message. Then Peer1 waits for Peer2. When Peer2 wants to connect, it sends a initialize message to the Server and becomes aware that it is not the session initiator. The Server then sends peerChannel and got user media for both Peer1 and Peer2.

After this exchange of messages, Peer1 and Peer2 can start to establish a P2P session. They first need to inform each other of the media type, format, codecs, and all the other properties that will be used in the session. This information will be in the “offer” and “answer” messages that are transmitted using SDP. Peer1 sends the “offer” to the Server, and this one sends it to the Peer2. Peer2 sends the “answer” to the Server that sends it to Peer1.

They also need to know in which will be the protocol used, and in which IP address and port the session will be established. For this ICE “candidates” messages will be sent. Again, the Server will intermediate this exchange of messages: Peer1 sends its “candidate” to the Server, this one sends it to the Peer2. Peer2 sends its “candidate” to the Server, this one sends it to the Peer1.
When the exchange of SDP and ICE messages is over, the connection is established and the peers can communicate without using the Server as an intermediary. Peers use their web browsers to communicate with each other and nothing else. There is no need to install any plugins.

In the experimental tests, the application worked perfectly and it could be concluded that the implementation of the project was a success. The authors also concluded that it is feasible to send other types of media and files with WebRTC and to encrypt them too, which is something very important, not as much as in a VPN, but more on public networks with a high risk of having their signals intercepted.

This project demonstrates that WebRTC is a powerful technology that can improve the overall communications, changing the traditional way media and files are exchanged.

### 3.4 P2P Live Video Streaming in WebRTC

The project described in [37] was developed to see if it would be possible to implement live video streaming into web applications, running on a browser, using WebRTC. It was developed in a controlled environment, that can be called a “laboratory”, and not tested in a real world situation.

Figure 3.5 illustrates the architecture of this project. The WebRTC Coordinator maintains the WebRTC network and, in this project, is also the source publisher. The PeerJS Server is a comple-
ment to the WebRTC Coordinator, supporting it in the establishment of connections between peers.

Figure 3.5: Overall architecture of P2P Live Video in WebRTC, adapted from [37]

The connection of peers is very simple. A user that wants to join the peer network registers at the PeerJS Server and at the WebRTC Coordinator, using a user ID, generated by itself. After, the WebRTC Coordinator will select, pseudo-randomly, two peers in the WebRTC network and tells the peer trying to connect who they are. The peer opens a RTCDataChannel to the two peers and becomes part of the WebRTC network.

If there are no peers in the network and a peer tries to connect to the WebRTC network, it will only subscribe to the source publisher. Then, the second will subscribe to the source publisher and the first peer. The third one will connect, pseudo-randomly, to two of the peers already in the network: the first peer, the second peer and the source publisher. A random mesh is created every time a new peer joins the WebRTC network.

One important detail of the implementation of this project is that every peer connects to exactly two other peers, with the exception of the first peer, obviously. This means that the peers that join the WebRTC network first are going to have more peers connected to them, because they are nearer the source publisher. Peers that stay in the network for long periods of time will have more peers connected to them, will be nearer the source publisher and will be more stable.

In the experimental tests, it was measured how well a file with 50 KB would be transmitted in a network in which the number of nodes varied between 2 and 32. The metrics used were the total packets sent, control packets received, redundant packets received and delivered packets received. The authors measured the metrics for these tests by determining the time that it took for each peer to receive the packet and the total number of packets received.

The tests showed a big discrepancy between the total packets and the combined value of control packets received, redundant packets received and delivered packets received, which can be explained by the fact that it was used UDP to transfer the data.
They also showed that a smaller buffer map interval means that the file reaches the nodes faster, but it also means that the control packets have a significantly bigger overhead. Because of this, it is necessary to have a compromise between overhead and delay, that will be the buffer map interval.

In conclusion, this project proves that it is possible to implement P2P live streaming using WebRTC in more complex scenarios, involving dozens of peers, which is already a considerable number, and achieve good enough quality. The only downside was the fact that, at the time of the realization of this project, WebRTC still had limitations that made this implementation unpractical, with the major one being the restrictions imposed by the browsers vendors, which did not permit to achieve a good performance, nor to have interoperability between different browsers.
Architecture of the Prototype

Contents

4.1 Requirements ............................................................... 35
4.2 System Design ............................................................. 36
4.3 System Components ..................................................... 39
This chapter describes the architecture of the proposed prototype solution. It starts by listing the requirements, followed by the presentation of the design of the system. Lastly, each module present in the architecture is briefly described, for a complete understanding of the system.

4.1 Requirements

This solution is intended to be applied to a multitude of distinct scenarios and systems, but always with the objective of implementing real-time collaboration in a P2P fashion focused on live video streaming, using only a simple web browser. Because of that, there are some requirements that need to be fulfilled independently of the scenario or system:

- **Allow for P2P collaboration**: the main objective of the project is to allow for peers to collaborate with each other, with video streaming, in a P2P fashion using WebRTC;

- **Peers on the website can collaborate**: only peers in the same website, watching the same video stream, can collaborate with each other. It is considered that they belong to the same swarm;

- **All peers can contribute**: all peers that use the system can contribute with an original stream of their own. Nevertheless, a peer that wants to collaborate with their own live stream needs to have a webcam and a microphone, being this the only restriction;

- **Self-scalable**: the solution must be able to maintain Quality of Experience (QoE) and QoS no matter how many peers are using it;

- **Stream of video**: system needs to provide to the peers the ability to stream video and to receive the content they want in a P2P fashion;

- **Reduce Media Server entity load**: a Media Server entity in the system should be considered a super-peer containing the original contents available to users. The first peers connects to this super-peer to receive the desired content, but, after the P2P network (swarm) starts to compose, this server will start reducing, or stop having, peers connected to it, reducing its load;

- **Ability to interact with the system**: the peer should be able to interact with the system in order to fully take advantage of all the functions that it permits.

The core of this solution is the software that will be implemented, especially the web application as it is responsible for combining all technologies in charge for permitting to achieve real-time collaboration in a P2P fashion focused on live video streaming, and to let the user take advantage of them in an easy way. Relatively to the hardware, nothing out of the ordinary is used, mainly consisting in servers, and devices with access to the Internet.
4.2 System Design

Based on the related works studied and on the research work done, it was concluded that the best system architecture for the solution is the one illustrated in Figure 4.1. This architecture, consisting of a Peer Coordinator, Web Server and Media Server, a Signaling Server, and an arbitrary number of Peers, is designed to be easily integrable with any web stack in the Internet.

To be easy to illustrate and explain, the system is shown only with three peers, but, like aforementioned, it is designed to be able to service as many peers as necessary. These three peers represent a swarm. The Peer Coordinator in this solution, that works like a tracker, does not communicate with other Peer Coordinators.

This means, in practice, that the peers need to directly access the Peer Coordinator responsible for this swarm to be able to join this P2P network and access the video streams. The web application is not “represented” in the architecture diagram, as it is just the browser RTCWeb process running in the peers.

It should be noted that although the Peer Coordinator, Web Server and Media Server are represented in the same physical node, they are in fact different modules (entities), each with a different function. Also, the Media Server module will act as a peer, a super-peer, that will distribute the content to the first peers that request it. Afterwards, the Media Server will no longer be requested by the peers who arrive later, as long as there are enough peers (at least one) with the whole content in the swarm.

The peer node process in the browser consists of two very important modules: the Content Loader and the Connection API. The Content Loader handles everything related with the video stream and decides from which peer or peers the stream should be requested from. The Connection API handles the connections between the peers.
In Figure 4.2 it can be observed the interactions between the various participants in the system. This flow diagram only represents the interaction of two peers, but the interaction for more peers is linear and can be perceived from the diagram. The flow presupposes that the system is started at the beginning of it and that there are no network errors during it. The interaction is described as follows:

- Peer 1 accesses the website and receives the content of the website.
- Peer 1 registers itself with the Peer Coordinator and receives a confirmation of its registration.
- Peer 1 then requests the content it wants to the Content Loader that signals the Connection API that Peer 1 wants that Content.
- The Connection API requests the list of peers that have the Content to the Peer Coordinator.
- The Peer Coordinator sends the list to the Connection API, that sends it to the Content Loader.
- The Content Loader decides to get the Content from the Media Server, the super-peer, due to the fact that Peer 1 is the first user to request the content.
- The Content Loader tells the Connection API to get the Content from the Media Server and a connection is established.
- A bidirectional data channel is established between Peer 1 and the super-peer, the Media Server.
- Finally, the Connection API will send the Content that is receiving to the Content Loader, that displays it to the peer, and announces to the Peer Coordinator that it is now a source for the Content.
- Peer 2 will request the content to Peer 1 instead of requesting it to the Media Server.
- The Connection API of Peer 2 will signal the Signaling Server, to gather ICE candidates and, after gathering them, will offer a SDP to the Peer Coordinator.
- The Peer Coordinator will transmit to the Peer 1 the SDP that was offered by the Peer 1.
- Then, the Peer 1 will also gather its ICE candidates and answer the SDP of the Peer 2, sending a SDP to the Peer Coordinator, that will send it to the Peer 2.
- A bidirectional WebRTC data channel is established between Peer 1 and Peer 2, and, finally, the Connection API will send the content that is receiving to the Content Loader, that displays it to the peer, and announces to the Peer Coordinator that it is now a source for the content.
The second interaction, represented in Figure 4.3 involves a peer wanting to collaborate with an original stream. This original stream can be only of devices from the peer or a mix between a device of the peer and another stream that the peer is watching.

It is assumed that the interaction begins after the peers are already in the P2P network (swarm) and no network errors occur during the interaction. The following steps describe the interactions:

- Peer 1 informs its Content Loader that wants to stream video. Then the Content Loader, in turn, informs the Connection API.

- The Connection API asks authorization to the Peer Coordinator, to stream video.

- After receiving authorization it informs the Content Loader, which will then send the stream to the Connection API.

- Peer 1 starts to stream and sends the stream to the Connection API, which in turn will announce, to the Peer Coordinator, that Peer 1 is the source of that stream;
Finally, the Peer Coordinator sends an update to the Web Server with the streams that appear in the website, in order for peers to be able to see that there is a new video stream available.

Another important aspect to consider when implementing the system is how a user will be registered. In this solution, the Peer Coordinator grants an ID to a peer that registers. This ID is used to give the user anonymity and will also be used in the list that the Peer Coordinator compiles of the peers that have a stream.

Also, a peer that starts to stream will not connect directly, in a P2P fashion, to all the other peers that want to see that stream. Instead, the peers that want to see the stream connect to one another in a random form. This implies that at least one of the peers is directly connected to the peer that is streaming. This will also mean that some peers might have some delay regarding peers nearer the original peer.

4.3 System Components

This section describes the components of the system. There are server components, peer components, the Signaling Server, which will be described but will not be part of the system, for simplification purposes, and the web application.

4.3.1 Server Components

This serving components will consist of three modules with distinct functions and functionalities: the Peer Coordinator, the Web Server, and the Media Server. These modules, which for the prototype are implemented in a single node, can be implemented in three different nodes if necessary.
The main objective of these server modules represented in fig. 4.4 to always provide the most up to date video and to control the network of peers.

Peer Coordinator: In a simple way, the Peer Coordinator will be the responsible for registering the peers, to give the authorization to the peers to collaborate with a video stream, to keep an updated list of streamers and sources, and to, in a certain way, assume the role of a tracker.

The registration of a peer is processed when it reaches the website. It sends a request to the Peer Coordinator and the Peer Coordinator will give it an ID. A peer ID only lasts for the time the peer is on the website. After leaving the website, and when returning, the peer needs to register again and is given a new ID. Also, the ID is a way to give the peer some anonymity during the process of establishing a WebRTC data channel.

As soon as a peer registers with the Peer Coordinator three tables are created. One table has all the connecting information needed such as the peers IDs and their IP addresses that will be used for the other peers to connect to each other. Each time a peer registers with the Peer Coordinator an entry for that peer is added to this table.

The second table maps the existing files that can be shared between the peers with the actual peers that have them. When a peer becomes the source of a content and announces it to the Peer Coordinator it is added to the list of peers that exists on this table. When a peer exits the website or does not respond to any message during some time or no longer has the content it is removed from the table.

A simple version of this is represented in Table 4.1, where each content has an unique ID, a hash, to distinguish them from the other content, and the peers are represented by a name, which is in fact the ID given to each peer when registered with the Peer Coordinator.

Table 4.1: Table of association of streams and their sources

<table>
<thead>
<tr>
<th>Content</th>
<th>Content Hash</th>
<th>Peer</th>
</tr>
</thead>
<tbody>
<tr>
<td>content1</td>
<td>gJiCLJxU27O2</td>
<td>[Peer2], [Peer3]</td>
</tr>
<tr>
<td>content2</td>
<td>hHePboV2z9c</td>
<td>[Peer1]</td>
</tr>
<tr>
<td>content3</td>
<td>xJqWGewQgum</td>
<td>[Peer3]</td>
</tr>
<tr>
<td>content4</td>
<td>WdorWWfCqgh4</td>
<td>[ ]</td>
</tr>
</tbody>
</table>

The third table, is the reverse table, that helps remove peers from the network. This table maps all the peers present in the network with the resources they have, in this case, all the content they are
source of. An entry in this table is created with an empty list when a peer registers with the Peer Coordinator. When the peer announces that it is a source of a content it is added to the list. When a peer exits the network, the Peer Coordinator removes the peer as a source for all contents and then removes the entry from the table.

These three tables are synchronized within the Peer Coordinator and are updated as the peers register, get out of the network, when they become the source of a stream or when they stop to be the source of a content.

Although in the solution only the IP and the ID are used, more extended information can be added to improve the quality of the P2P network and to deliver the streams in a more efficient way. Information such as the geographical position or the peers bandwidth can be used to improve the QoS.

The Peer Coordinator also works as an intermediary for passing the SDP from a peer to another peer. This is due to the fact that, when a peer receives the list of peers with the content, it does not receive the IP address of each peer, but only the ID of the peer, as seen in Table 4.1.

In a similar way, when a peer wants to collaborate with its own content, i.e., when a peer starts to stream, tables are created with the IDs of peers that are streaming and the IDs of peers that are watching and, because of that, are sources of those streams.

**Web Server:** The Web Server will host the website of this service, like mentioned before. It is a simple server, with no special characteristics, that distributes HTML and it directs the peers to the Peer Coordinator, for them to be registered.

It also provides a graphical interface to the peers in which the content is displayed, where they can collaborate with their own live video and where they can see which streams are available. Basically, the Web Server provides the peers the option to fully interact with the system and all its functionalities.

**Media Server:** This last module is essential to the solution. Although this module is described as a super-peer throughout this document, it is nothing more than a normal server which will distribute the content to the peers that connect to it. It will serve the content through a normal HTTP connection to the clients and will always be online, so that all peers have at least one stable source for the content provided by this server.

Due to the fact that in the beginning there are no peers, a certain number of peers will connect directly to it. When this number of connections is reached, the next peers to join the swarm will only connect to other peers and not to the Media Server, if there are peers with the full content they want. For the prototype implementation only the first peer connects directly to the Media Server and the peers to join after the first will connect to this first or other peers, as long as there are peers with the requested content.
4.3.2 Peer Components

Peers are users that not only download content but also upload it to other peers that want that same content. Peers can also collaborate with their own live video stream or to mix a stream of another peer with their own live audio stream. In this solution all peers can collaborate with their own live video stream, as they are considered reliable. When a peer collaborates with a live video stream it will appear with an ID on the website so the other peers have the option to see the new stream. This ID will appear to the peers when they refresh the list that has the IDs of the streams.

Two modules compose the peer process as illustrated in Figure 4.5: the Connection API and the Content Loader. The Connection API simplifies the WebRTC functionalities regarding the bidirectional data channel. The Content Loader takes decisions regarding the content, the video, and cooperates with the Connection API when it is required.

Figure 4.5: Peer modules

These two modules will be executable in the browser and are described as follows.

Connection API: For this solution to be fully implemented, it is necessary to have a module that communicates with the Peer Coordinator and the Signaling Server, and that transforms the creation of the WebRTC data channel between two peers in an easy to use function.

To connect to another peer the Connection API will start by selecting a STUN server to use. It is noteworthy to mention that the IP address resolution should be done by the Signaling Server, but in the implementation of this prototype a Google's public STUN server was used.

In case a STUN server can not resolve the address of a peer, there will be left only two options: use a TURN server or transmit an error message to the peer. Using a TURN server implies to have some costs and also will not provide a direct connection between the two peers. So, the second option will be used, the transmission of an error message to the peer, in case the IP address resolution fails.

After the STUN server resolves the IP address, the next step is to send the SDP to the Peer Coordinator which will send it to the intended peer. SDP is a set of rules that defines how multimedia sessions can be set up to allow all end points to effectively participate in the session. When the other
peer answers positively to the SDP it received the two peers can connect directly through a WebRTC data channel.

Through this newly created data channel the peers can send and receive video from each other.

**Content Loader:** The Content Loader will determine if a resource is already loaded, where it is going to be loaded from, and coordinates with the Connection API to open and manage the peer connections. In this solution the Content Loader loads content of and from the website and video streams from one or more peers. It will also interact with the local device, in case a peer wants to contribute with an original stream.

After a connection with other peer is made, there will be a continued stream of content and/or a video stream arriving. The module will reassemble it and, after the content and/or video is reassembled, it is displayed to the peer.

There is no mechanism to assume the correctness and authenticity of the content and/or video. It is assumed that all peers using the system, and that can collaborate with a stream, are trustworthy.

After starting to download the content or after starting to watch the stream, the Content Loader notifies the Peer Coordinator that this peer is now a source of that stream.

When a peer can not connect to other peers it will request the content from the media server. In case a peer can not connect to other peer that is streaming there are no mechanisms to avoid failure. The peer will have to try to reconnect to the peer that is streaming until it is successful.

When a peer closes the browser window, the listener will warn the Peer Coordinator to remove the peer from the list of sources. In the event of the browser or the computer crashing or the connection having problems the peer is removed from said list if it does not respond to any messages for more than a predefined time. For the prototype, that predefined time was set as 5 seconds.

### 4.3.3 Signaling Server

The Signaling Server will provide the peers with ICE candidates in order to be possible for them to establish connections between them. ICE is used to cope with NAT and firewalls, so that, no matter where a peer is, it can always connect to other peers.

Peers receive, in the ICE candidate, information about the IP address and the port to be used for the exchange of data, among other important information. This information is then transmitted, through the SDP, to other peers interested in establishing a connection between each other.

Although this Signaling Server is represented in the architecture, in this project a public signaling server provided by Google was used.
4.3.4 Web Application

In this prototype it is necessary to have a web application that provides the users with a Graphical User Interface (GUI) in which they can interact with the system and use all of its functionalities.

This web application in the prototype is very simple in terms of user interface, just for demonstration and testing purposes, but allowing the users to take advantage of all the functionalities, and leaving space to be improved and adapted as necessary.
This chapter gives an explanation of how the prototype solution was implemented. It begins by explaining the decisions taken relatively to the programming language and the development environment, before detailing how each component of the architecture was implemented. Lastly, it also gives an explanation of the problems encountered during the implementation phase and how, and if, they were resolved.

5.1 Implementation Options

To implement the prototype it was necessary to have a network of peers and to have all the components described in Chapter 4. Everything could be built from scratch, but that would take too much time and would lead to an effort beyond what would be necessary. For this reason the decision was made to use tools that would abstract and simplify this process.

These tools, due to the fact of being open-source, can be modified to suit the needs of the prototype, making them a good option for implementing it. One of the most important tools is the PeerJS. PeerJS is an open-source WebRTC API that wraps the implementation of WebRTC on the browser in an easy-to-use and configurable P2P connection API. PeerJS also has a connection broker, of optional use, that helps connecting the peers to each other.

As the objective of this implementation is to verify the feasibility of a real-time collaboration system in a P2P mode using live video stream, the use of PeerJS to abstract the WebRTC core functions is acceptable and will save a lot of programming work. Also, the PeerJS Server allows the extension of its functionalities, making it adaptable to implement the requirements necessary for this solution.

Although the Media Server present in this prototype solution works as a peer, it is still a module of a normal server, so the traffic to it should be kept to a minimum. The content provided by the Media Server will be served to the first peer that enters the website. Only the first peer in the P2P network will connect to the Media Server. It can be allowed to have more peers connecting to the Media Server, if found necessary or to improve the QoS.

When there is more than one peer with the content it is randomly chosen the peer that will transmit the content to the peer requesting it. As already stated, for the prototype the Media Server is excluded from this choice. If a peer cannot connect to other peers or if there are no other peers with the content it will, obviously, request it from the Media Server.

The content distributed by the Media Server is unique for each content. In this prototype this ID was defined by the content Uniform Resource Locator (URL). This ID is a hash generated based on the path where the content is stored and the Peer Coordinator with that ID, so that a peer can request it.

Because the work for this thesis is focused on real-time collaboration, this prototype allows peers to collaborate with their own live video stream or to mix a live video stream they are watching with their live audio. Other characteristics may be added to the prototype in the future due to the fact of the software
used being easy to extend in terms of functionalities.

When a peer starts to collaborate, i.e., starts to stream, its ID is stored in the Peer Coordinator and will be displayed on the main page website, so that every peer that accesses the website can watch that stream if they desire. When a peer starts watching a stream it becomes a source of that stream. This is meant to divide the load by many peers and not overload the peer that is streaming.

In this prototype it is assumed that all peers are good peers, i.e., they are not malicious, do not send virus, spam, malware, or any other type of data that can put in risk the other peers. For this reason, no verification mechanism of the content integrity is executed.

For the purpose of testing, due to the fact that the connection broker is an Express.js server working with Node.js, it is easier to create an Express.js server that makes the video available and paths reserved for PeerJS and WebRTC related actions.

**PeerJS**: PeerJS wraps the implementation of WebRTC on the browser in an easy-to-use and configurable P2P connection API. Nothing else but an ID is necessary for a peer to create a P2P data or media stream connection to another peer. This ID, which is a string, can be chosen by the peer itself or is assigned by the server, that generates a random ID.

It is necessary a server to act as a connection broker to actually connect the peers to each other. This server is provided by PeerJS, with the PeerJS Server, an open-source implementation of this connection broker, written in Node.js. The PeerJS Server is only needed for connecting the peers to each other. After that, any communication happens directly between the peers.

Since PeerJS and the PeerJS Server were developed in JavaScript, the language used for the implementation of the project will be JavaScript, in order to facilitate any modifications necessary.

**Node.js**: Node.js is an open-source, cross-platform runtime environment for developing server-side applications and to build scalable network applications. It is a JavaScript runtime built on Chrome's V8 JavaScript engine, making code execution very fast.

It uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, making it perfect for data-intensive real-time applications that run across distributed devices. Node.js package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

Even though Node.js is not a JavaScript framework developers can write new modules in that programming language, because many of the basic modules of Node.js are written in it.

**Express.js**: Express.js is a web application framework for Node.js that provides a robust set of features for web and mobile applications, making it easy for developers to build web applications. By using Express.js there will be no need to have two servers running separately in order to serve the static files, in this case, a video.
5.2 Implementation of Components

In this section the most important parts of the implementation regarding the programming will be described for a better comprehension of the methods and functions used in the prototype. The section is divided into three parts, the server components, the peer components, and web application, each with excerpts of the most important methods and functions used in the prototype.

5.2.1 Server Components

The Web Server and the Media Server will be implemented in the same node. The content that will be provided by the Media Server will be a recorded video. The video will be a file, that will be automatically transmitted to the first peer that enters the website. The other peers will also automatically request and receive the video but in a P2P fashion from peers that already have the video.

**Media Server:** The Media Server component is nothing more than a normal HTTP server that sends the video to the clients that request it. For this, a simple line of code, as illustrated in Listing 5.1, was enough for serving static files in a directory, a video in this case.

**Listing 5.1:** Code for the server HTTP to serve all the files in the public directory

```javascript
app.use(express.static(__dirname + '/public', { etag: false, lastModified: false }));
```

**Web Server:** The Web Server coexists in the same node as the Media Server. For this reason, the code of the Web Server and Media Server is one and the same, with the code for serving static files being shown in Listing 5.1.

In Listing 5.2 it is explicit the code for the Web Server, and, as consequence, for the Media Server, to be able to listen to requests from the various peers that access the server.

**Listing 5.2:** Code for the web server

```javascript
var server = app.listen(8000, function () {
    var host = server.address().address;
    var port = server.address().port;
});
```
**Peer Coordinator:** For the Peer Coordinator a Representational State Transfer (REST) architecture is used and each path will lead to an action on the Peer Coordinator side. First it is created, on the HTTP server, a path, /peerjs, and everything that accesses that path will be handled by the Peer Coordinator. Additionally, the peers need to be able to make peer list requests, so the discovery flag was allowed.

Listing 5.3: Code for the coordinator with a reserved path

```javascript
var options = {
    allowDiscovery: true,
    allow_discovery: true,
};
app.use('/peerjs', ExpressPeerServer(server, options));
```

Eight extra functionalities must also be added to the connection broker, for the requirements established for the Peer Coordinator to be met (see Table 5.1):

- Announce peer as a source of content;
- Get list of peers with content;
- Announce peer as a main streamer;
- Get list of peers that are main streamers;
- Delete a peer from the list of peers that are main streamers;
- Announce peer as a secondary streamer;
- Get list of peers that are secondary streamers;
- Delete a peer from the list of peers that are secondary streamers.

A peer can announce itself as a source of content and/or as a streamer and/or a secondary streamer. For announcing itself as the source of content, a peer sends the request /content/arsxmn10p/6db22aff to the Peer Coordinator.

This means that the peer, with the ID arsxmn10p, is a source of the content with the hash 6db22aff. The same is done for a peer to announce itself as a streamer; send the request /streamer/arsxmn10p/Nn2vix14 to the Peer Coordinator. This means that the peer with the ID arsxmn10p is live streaming and other peers can connect to the stream and see it. The hash Nn2vix14 is equal to all streamers and works as a flag to make it easier to obtain the list of streamers. It is worth to note that a main streamer is also a secondary streamer, so that peers can also connect to it and not only to pure secondary streamers.
Table 5.1: Functionalities added to the Peer Coordinator

<table>
<thead>
<tr>
<th>Path</th>
<th>Function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>/content/:id/:contenthash</td>
<td>Announces a peer with :id as source of content with :contenthash</td>
<td>:id - ID of the peer that is announcing :contenthash</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Hash of the content</td>
</tr>
<tr>
<td>/peers/:contenthash</td>
<td>Lists all peers that have the content with :contenthash</td>
<td>:contenthash - Hash of the content</td>
</tr>
<tr>
<td>/streamer/:id/:streamhash</td>
<td>Announces a peer with :id that is now a main streamer</td>
<td>:id - ID of the peer that is announcing :streamhash</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Hash to differentiate the main streamers</td>
</tr>
<tr>
<td>/streams/:streamhash</td>
<td>Lists all peers that are streaming</td>
<td>:streamhash - Hash to differentiate the main streamers</td>
</tr>
<tr>
<td>/deletestreamer/:id/:streamhash</td>
<td>Deletes the peer with :id from the list of main streamers</td>
<td>:id - ID of the peer to be deleted :streamhash</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Hash to differentiate the main streamers</td>
</tr>
<tr>
<td>/secondarystreamer/:id/:secondarystreamhash</td>
<td>Announces a peer with :id that is now a secondary streamer</td>
<td>:id - ID of the peer that is announcing :secondarystreamhash - Hash to differentiate the secondary streamers</td>
</tr>
<tr>
<td>/deletesecondarystreamer/:id/:secondarystreamhash</td>
<td>Deletes the peer with :id from the list of secondary streamers</td>
<td>:id - ID of the peer to be deleted :secondarystreamhash - Hash to differentiate the secondary streamers</td>
</tr>
<tr>
<td>/secondarystreams/:secondarystreamhash</td>
<td>Lists all peers that are secondary streamers</td>
<td>:secondarystreamhash - Hash to differentiate the secondary streamers</td>
</tr>
</tbody>
</table>

A secondary streamer is a peer that is watching a stream and can transmit that stream to other peers, to relieve the main streamer in case there are a lot of peers watching the same stream. The peer that will be a secondary streamer sends the request /secondarystreamer/bteqWc4Mo/gI0Zva42, with bteqWc4Mo being the ID of the peer and gI0Zva42 being a hash based on the ID of the peer that is the main streamer.

When a peer stops streaming, it needs to be deleted from the streamers list. For this, when the peer decides to stop streaming, a request is sent to the server for that peer to be deleted; deletestreamer/arsxm10p/Nn2vix14 is sent to the server, with arsxm10p being the ID of the peer to be deleted from the list and Nn2vix14 being the flag that signalizes the streamers.

As well, when a secondary streamer stops watching that specific stream or disconnects it sends a request, to the server, to be deleted from the list of secondary streamers; deletesecondarystreamer/bteqWc4Mo/gI0Zva42, where bteqWc4Mo is the ID of the peer to be deleted from the list and gI0Zva42 is the hash based on the ID of the peer from which the secondary streamer was watching the stream.
In all cases, the response does not need to be interpreted.
Peers can obtain the list of peers with content or that are streaming with an HTTP GET with the path `/peers/6db22aff`, with '6db22aff' being the hash of the content requested, or with the path `/streams/Nn2vix14`, with 'Nn2vix14' being a hash to differentiate the streamers, respectively.
For obtaining the list of secondary streamers the procedure is identical, with an HTTP GET with the path `/secondarystreams/gI0Zva42`, where 'gI0Zva42' is the hash based on the ID of the peer that is streaming.
As responses, the functions will return a list with IDs of the peers that have the content or a list with IDs of the peers that are streaming.

5.2.2 Peer Components

The Peer Components are essentially the Connection API and the Content Loader. The Connection API sends requests to the serving components, handle errors and returns answers. The Content Loader enables the P2P video file request and the real-time collaboration.

**Connection API:** To briefly describe what the Connection API does, it sends an HTTP request to the /peerjs, handles errors that may occur and returns an answer. The extra functionalities added to the Peer Coordinator must be available to the clients, so the PeerJS client script has to be extended to handle them.
The Content Loader needs to access the two functions shown in Listings 5.4 and 5.5 to be able to implement the extra functionalities.

**Listing 5.4:** Code for listing all peers with a certain content

```javascript
peer.listAllPeersWithContent(contenthash, callback);
```

**Listing 5.5:** Code for listing all peers that are secondary streamers

```javascript
peer.listAllSecondaryStreamers(streamhash, callback);
```

If an error occurs or there is no response an empty list is returned in all cases. For the first function, the peer will get the content directly from the HTTP, so there are no serious consequences. For the second function the peer will not be able to know to which peers they should connect to to receive the stream they want, so the collaboration part of the implementation is affected.
Content Loader: The Content Loader can be divided in two parts: the P2P video file and the real-time collaboration. These two parts are put together in a JavaScript script that is executed after the Document Object Model (DOM) is loaded. The two parts will be described separately, with the P2P video file being described first and the real-time collaboration after.

The first action of the Content Loader is to register the peer with the Peer Coordinator, seen in Listing 5.6, obtaining a random ID. Host can be an IP, a URL or blank to use the same host as the HTTP server.

Listing 5.6: Code for registering the peer with the Peer Coordinator

```javascript
var peer = new Peer({
  host: 'x.x.x.x',
  port: 80,
  path: '/peerjs',
});
```

Concerning the P2P video file, after the objects in the HTML are accessible the script will collect information about the video. To prevent the video to always be retrieved from the HTTP server the `src` attribute has to be replaced by a `data-src` attribute.

Listing 5.7: Video selector attribute

```html
<video id= "mainvideo" data-src="/vid/30.mp4" allowfullscreen type="video/mp4" controls> </video>
```

It was decided that when determining what resources should be requested the browser cache will be ignored. This is not ideal, because a local request is always better than a network request, but it is done for testing purposes.

To request the video to a peer it is generated the video’s hash, based on its URL, and then the request is sent to the Peer Coordinator, as can be seen in Listing A.1. The peer that made the request obtains a list with all the peers that can share the video and will download the video from one of them. In case there are no peers with the video or an error occurs the video is requested from the HTTP server.

It is necessary to configure the connection in case any request is received by the peer. This is fundamental because it is expected that JavaScript behaves asynchronously when dealing with web requests.
Peers need to always be ready to receive the content that they requested or to receive a request to
the content of other peer. When the peer receives the video that requested it will need to process it
and integrate it in the DOM to be shown. If the peer receives a request of another peer, and has the
video, it will send the video through the DataChannel.

As can be observed in Listing A.2 the peer will try to connect to the peer it has chose to receive the
video from. If the connection is opened the peer sends the request for the video with the video’s
hash.

Also in Listing A.2 it can be observed that the peer tries to connect three times in a 500 milliseconds
interval and if it is not able, will retrieve the video from the HTTP server, thus guaranteeing an
alternative way to receive the video.

After a peer receives the video, this is saved in a structure of easy access. This way, in case another
peer requests the video, that request can be answered more rapidly, improving the QoS.

Due to the fact that the local storage of the browser has a very low limit, it is used a local variable,
as seen in Listing 5.8. The disadvantage of this method is that it consumes more resources from the
client side.

### Listing 5.8: Example of video in local storage

```plaintext
L3ZpZGVvLzc2Vjb25kGltZXIubXAO {
  id: "L3ZpZGVvLzc2Vjb25kGltZXIubXAO",
  datasrc: "/video/30secondtimer.mp4",
  hash: "L3ZpZGVvLzc2Vjb25kGltZXIubXAO",
  fullurl: "/video/30secondtimer.mp4",
  content: "data:video/mp4;base64,
AAAAGGZ0eXBtcDQyAAAAGlzb21tcDQyAABBzG1vb3YAAABsbXZoZAAAAADT0...kT/
Coe+z/hDikKv7VlnEx+Zv/yH707dw6Jr4N",
}
```

The content of the video is saved in base64 to be sent immediately after a request, without losing
any time in preparing the data.

For the collaboration part, the peer that is streaming will be defined as the main streamer and also
as a secondary streamer. If a peer starts watching that stream, that peer will be only defined as
a secondary streamer. New peers that want to see the stream will randomly connect to the main
streamer or to a secondary streamer. This is done so that not all peers connect to the main streamer
and do not overflow the main streamer.

This will mean that both the main streamers and the secondary will need to alway be prepared to
receive and automatically answer any calls from the other peers, to provide a fluent collaboration with
minimal effort for the main and secondary streamers.

It is to note that a peer can not be a secondary streamer of a stream besides its own if he is streaming.
This is, even if a peer is watching a stream while streaming, the peer will only relay the original stream
and not the remote stream to other peers.
All calls a peer receives as main streamer or as secondary streamer are placed in lists so that if the peer stops streaming all calls are disconnected or if it stops watching a determined stream all calls are disconnected.

A special case occurs when a peer is streaming and, at the same time, watching a stream from another peer. In this case, if that peer receives a call it will answer that call with a mix of streams: video from the stream it is watching and its own audio. The peer that made the call will receive that mix of streams like it is only one stream, as can be seen in Listing A.3. This is done in an automatic way and with those specific streams in order to be evaluated later.

At any moment a peer can decide to become a streamer, needing only to press the corresponding button on the interface of the application. It is then asked authorization to access the peer’s webcam and microphone and the peer stops being a secondary streamer of a stream in case it was one. Only if the peer authorizes the access to its hardware becomes a streamer.

Then it is created a local stream and this is shown to the peer. Lastly, the peer that is streaming is added to the list of main streamers and also to the list of secondary streamers of its own stream, as can be seen in Listing A.4.

Obviously, a peer can also stop being a streamer at any time by pressing the button to stop the streaming. This will cause the peer to revoke the authorization that it gave to share its webcam and microphone, stopping the local stream, and all peers that were connected to the peer will be disconnected from it.

The peer will also be deleted from the list of main streamers that are available to all peers and of the list of secondary peers of its own stream.

Also, in case the peer is watching a stream from one other peer, it will be added to the list of secondary streamers from that stream, as can be observed in Listing A.5.

For connecting to a stream, the peer that wants to connect to that stream, in case the peer was watching other stream, will start by being removed as a secondary streamer of the first stream. After, it will be requested the list with all the secondary streamers for the stream that the peer wants to watch in that moment.

Then the peer will connect to the main streamer or a secondary streamer of the desired stream, in a random way. For this a request is placed to the main or to a secondary streamer for a connection and, if it is successful, the stream will be shown to the peer that wanted to see it.

The call will also be placed in a list, for when a peer wants to stop seeing a stream that call can be correctly disconnected. Finally, the peer is added to the list of secondary streamers of the stream that is watching, as it can be seen in Listing A.6.

As expected, a peer can also stop watching the stream that is being displayed by pressing the correspondent button. This will close the connections between the peer watching the stream and
the streamer and the peers connected to this peer through the secondary streamer functionality. If a peer wants to watch the same stream it needs to repeat what is done in Listing A.7. To see all the peers that are streaming, e.g., that are main streamers, it is only necessary to press a button and a list with all the main streamers will be shown to the peer. The peer will need to always press a button for the list to be updated, there is not an automatic actualization of the list while new peers start streaming and others stop streaming, as it can be seen in Listing A.8.

5.2.3 Web Application

The web application will allow the peers to interact with the system, by giving them methods to stream video, watch streams, among others, through a GUI. It is a very simple interface to make it as intuitive as possible. HTML, Cascading Style Sheet (CSS) and jQuery were used to implement it.

Represented in Figure 5.1 is the main page of the web application. Peers will access a web page that will automatically play the video, and that gives them the possibility to interact and use all the functionalities of the system.

![Figure 5.1: Interface of the main page of the web application](image)

In Figure 5.2 is possible to observe how the interface will look like after a peer becomes a streamer. The stream has the peer's webcam and microphone as sources.

Figure 5.3 shows how the interface will behave when a peer is watching a stream. The main video, that can be seen in Figure 5.1, will disappear and the stream that is being watched will appear in its place.
Figure 5.2: Interface of the web application when streaming

Figure 5.3: Interface of the web application when watching a stream
6 Evaluation

Contents

6.1 Tests Objectives ................................................. 61
6.2 Deployment and Hardware ....................................... 61
6.3 Tests Scenarios ..................................................... 61
6.4 Evaluation of Results ............................................. 65
This chapter presents the objectives of the tests that were performed on the prototype to evaluate if it fulfills the goals proposed. It also presents all the results of those tests, the analysis of the results and draws some conclusions about the feasibility of the prototype and the contributions of this thesis.

6.1 Tests Objectives

The tests performed have the objective of measuring the resources that are consumed by the users, namely, the Random Access Memory (RAM) and Central Processing Unit (CPU) usage, and also the quality of the transmission, by measuring the latency of both video and audio, and the Frames Per Second (FPS) of the video.

These performance and quality tests will demonstrate that the prototype developed is stable and that the QoS is guaranteed at all times.

6.2 Deployment and Hardware

The prototype was tested in a controlled environment, so that the conditions in which it was tested can be easily configured, and to assess that the prototype works as it was supposed to, before deploying it to the Internet and WWW.

For this reason it was decided to deploy the system in virtual machines. The server machine configuration had 256 MB of RAM and run Ubuntu Operating System (OS) in headerless mode. Node.js was used as the software for the prototype’s server.

Users, or peers, used for the tests were also running in virtual machines, but with GUI (desktop mode). Every peer was configured with 512 MB of RAM and running Ubuntu OS, with Firefox installed to access the website.

6.3 Tests Scenarios

All the tests scenarios were repeated three times, each time having a duration of 15 minutes. This allowed to achieve a better understanding of how the prototype behaved.

After each test, the measurements, captured locally by each peer, were recorded and then manually processed.
6.3.1 Test Scenario 1

The first scenario is very simple (Figure 6.1). In this interaction, Peer 1 starts by registering and then downloads the video file from the Server. Then it will start its own stream.

Peer 2 will register with the Server and download the video file from Peer 1. After that, Peer 2 will start to visualize the stream of Peer 1 and it will continue to visualize it for 15 minutes.

This scenario intends to demonstrate the most simple interaction between two peers and to measure how good is the performance to have a comparison in the other scenarios.

6.3.2 Test Scenario 2

The second scenario is an extension of the first scenario, but instead of only two peers there will be five peers (Figure 6.2). Peer 1 will be the first peer to enter the swarm, will register with the Server and download the video file. After that, it will start its own stream.

Peer 2, Peer 3, Peer 4 and Peer 5 will enter the swarm later, register with the Server and then download the video file from a peer instead of the Server. After that they will start to visualize the stream of Peer 1 but will not connect directly to Peer 1. Instead, they will connect to one another in a sequentially way, as it can be observed in Figure 6.2.

The goal of this scenario is to assess how the prototype performance changes with the increase in the number of peers and to have an estimate of the delay a peer can suffer.
6.3.3 Test Scenario 3

The third scenario involves only three peers. As shown in Figure 6.3 there are two streams, one represented by the red arrow and another represented by the blue arrow.

In this scenario Peer 1 will be the first peer to enter the swarm, will register with the Server and download the video file. After that it will start its own stream.

Then Peer 2 will enter the swarm, register with the Server and download the video file from Peer 1. After that Peer 2 will start to visualize the stream of Peer 1 for 5 minutes, when it will start its own stream. This stream will be a mix of the video stream that Peer 2 receives from Peer 1 and the audio stream of Peer 2 itself. This stream will be the one sent to Peer 3.

Finally, Peer 3 will enter the swarm, registers with the Server and downloads the video file from one
of the peers, and then will start to visualize the stream of Peer 2 for the rest of the duration of the test scenario.

The objective of this scenario is to measure how much delay there would be when streams with two different sources are mixed and sent to another peer and have a comparison for more complex scenarios.

6.3.4 Test Scenario 4

The fourth, and last, scenario (Figure 6.4) is identical to the third scenario but with more peers in the swarm. Again, there are two different streams, identified by the red and blue arrows, respectively.

![Figure 6.4: Diagram of Test Scenario 4](image)

Once more, Peer 1 will be the first peer to enter the swarm, will register with the Server and download the video file, and, after, it will start its own stream.

After, Peer 2 will enter the swarm, register with the Server and download the video file from Peer 1. Then Peer 2 will start to visualize the stream of Peer 1 for 5 minutes, when it will start its own stream. Like in the third scenario, this stream will be a mix of the video stream that Peer 2 receives from Peer 1 and the audio stream of Peer 2 itself. The stream will be sent to the other peers.

Lastly, Peer 3, Peer 4, Peer 5 and Peer 6 will enter the swarm, register with the Server and download the video file from a peer that has it. After this they will start, sequentially, to visualize the stream of Peer 2. Not all peers will connect directly to Peer 2, some of them will connect to peers that are just relaying the stream of Peer 2, as it can be seen in Figure 6.4.
The objective of this scenario is to assess how the prototype performance will behave with an increase of the number of peers and to have an estimate of the delay a peer can suffer when receiving a stream that is a mix of streams of two different sources.

6.4 Evaluation of Results

In order to arrive at a conclusion about the performance of the prototype it is necessary to have beforehand some reference figures for the parameters that were going to be measured.

For that reason, it was tested how much RAM and CPU Firefox consumed, when it is open on the homepage and in an idle state. This test was performed during a period of 15 minutes, with the values being registered every minute.

Concerning the RAM usage, the values fluctuate between 163 megabytes and 207 megabytes, considering all the tests. These will be the values of reference when testing the prototype (Figure 6.5).

![Figure 6.5: RAM used by Firefox in an idle state](image)

Due to the fact of Firefox being in an idle state, the usage of the CPU will not be too high, and that can be seen in Figure 6.6. These values will also be used as reference, but it is necessary to consider the fact aforementioned and expect an immense increase in the usage of the CPU when the prototype is being tested.

Considering the other parameters, latency of audio and the FPS of the video, some standard reference values were used. The latency, according to [38], should not surpass the 150 milliseconds. Latency depends on the network speed and conditions, which means that two users may experience very different latency values. Even though, any user should not have more than 150 milliseconds of latency to have a good enough QoS and QoE. The reference value of the minimum FPS acceptable are
24 FPS, according to [39].

![CPU Usage Graph](image)

**Figure 6.6:** CPU used by Firefox in an idle state

### 6.4.1 Results from Scenario 1

The measurements of the RAM, CPU, latency and FPS were done on Peer 2. This was due to the fact that Peer 2 was the one receiving video and audio, so it was important to see the performance of the prototype on this peer.

As can be seen in Figure 6.7, there was a little increase in the RAM used, as it was expected, with the values fluctuating between 263 megabytes and 327 megabytes, considering all three tests. This is an increase of around 58 percent, in both cases, in comparison to the idle state.

![RAM Usage Graph](image)

**Figure 6.7:** RAM used by Firefox in scenario 1
This increase can be considered normal, which demonstrates that the prototype, in this scenario, does not cause the web browser to consume too much RAM, making the prototype adequate to use integrated with a web browser.

With the use of the web application, it was more than expected that the use of the CPU would increase drastically, as it can be seen in Figure 6.8.

![CPU Usage](image)

**Figure 6.8:** CPU used by Firefox in scenario 1

While in an idle state the percentage of CPU used varied between 0 and 3 percent, during the use of the web application the values varied between 12 and 25 of percentage of CPU used, considering the three tests. The values can be considered normal, demonstrating that the prototype does not make the web browser to consume an abnormal percentage of the CPU.

Latency values can be observed in Table 6.1. Due to the fact of the network having very good conditions and having only two peers on the swarm the values are very low, which will make latency almost imperceptible to the users, leading to a very good QoS and user experience.

**Table 6.1:** Latency in scenario 1

<table>
<thead>
<tr>
<th>Minutes</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14 ms</td>
<td>19 ms</td>
<td>22 ms</td>
</tr>
<tr>
<td>2</td>
<td>20 ms</td>
<td>13 ms</td>
<td>18 ms</td>
</tr>
<tr>
<td>3</td>
<td>20 ms</td>
<td>13 ms</td>
<td>18 ms</td>
</tr>
<tr>
<td>4</td>
<td>13 ms</td>
<td>12 ms</td>
<td>18 ms</td>
</tr>
<tr>
<td>5</td>
<td>14 ms</td>
<td>16 ms</td>
<td>20 ms</td>
</tr>
<tr>
<td>6</td>
<td>16 ms</td>
<td>13 ms</td>
<td>22 ms</td>
</tr>
<tr>
<td>7</td>
<td>12 ms</td>
<td>15 ms</td>
<td>14 ms</td>
</tr>
<tr>
<td>8</td>
<td>13 ms</td>
<td>20 ms</td>
<td>17 ms</td>
</tr>
<tr>
<td>9</td>
<td>15 ms</td>
<td>13 ms</td>
<td>14 ms</td>
</tr>
<tr>
<td>10</td>
<td>20 ms</td>
<td>21 ms</td>
<td>13 ms</td>
</tr>
<tr>
<td>11</td>
<td>12 ms</td>
<td>14 ms</td>
<td>14 ms</td>
</tr>
<tr>
<td>12</td>
<td>13 ms</td>
<td>16 ms</td>
<td>22 ms</td>
</tr>
<tr>
<td>13</td>
<td>20 ms</td>
<td>14 ms</td>
<td>15 ms</td>
</tr>
<tr>
<td>14</td>
<td>14 ms</td>
<td>22 ms</td>
<td>12 ms</td>
</tr>
<tr>
<td>15</td>
<td>21 ms</td>
<td>16 ms</td>
<td>14 ms</td>
</tr>
</tbody>
</table>
Finally, relatively to the FPS of the video that the peer receives, the values can be seen in Table 6.2. Only the minimum, maximum and average FPS were measured during each test. In the first test there was a drop below of FPS the reference value of 24 FPS, but it is not a significant drop of FPS and will not affect in nothing the user’s experience.

<table>
<thead>
<tr>
<th>Table 6.2: FPS in scenario 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1</td>
</tr>
<tr>
<td>Minimum</td>
</tr>
<tr>
<td>Average</td>
</tr>
<tr>
<td>Maximum</td>
</tr>
</tbody>
</table>

Summary: In this test scenario, that represents the most basic interaction between two peers that the prototype allows, all the measurements made indicate that the prototype will deliver the quality expected of a system like this, even when deployed in a real world situation.

6.4.2 Results from Scenario 2

The measurements of the RAM, CPU, latency and FPS were done on Peer 5, because this was the last peer on the chain, meaning that it was more important to do the measurements in this peer to see how the prototype performs.

Values of the RAM used varied between 216 and 330 megabytes, when considering all three tests. This is an increase of 33 and 60 percent, respectively, from the minimum and maximum values of the RAM used in an idle state (Figure 6.9).

![Figure 6.9: RAM used by Firefox in scenario 2](image)

The increase is similar to the one registered in the Scenario 1, which can be deduced as a normal increase, in line with was suppose to happen when the web browser began to be used.
CPU values varied between 17 and 28 percent, considering all tests, as it can be seen in Table Figure 6.10. The values are similar to the ones in Scenario 1, which may be considered normal in these circumstances.

![CPU Usage](image)

**Figure 6.10:** CPU used by Firefox in scenario 2

Table 6.3 shows the latency values obtained in this scenario. As expected, they are very high, almost reaching 150 milliseconds. Even so, all the values are under the reference of 150 milliseconds, but still the latency may affect the user experience in certain situations.

**Table 6.3:** Latency in scenario 2

<table>
<thead>
<tr>
<th>Minutes</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100 ms</td>
<td>139 ms</td>
<td>136 ms</td>
</tr>
<tr>
<td>2</td>
<td>99 ms</td>
<td>102 ms</td>
<td>114 ms</td>
</tr>
<tr>
<td>3</td>
<td>130 ms</td>
<td>138 ms</td>
<td>102 ms</td>
</tr>
<tr>
<td>4</td>
<td>127 ms</td>
<td>124 ms</td>
<td>134 ms</td>
</tr>
<tr>
<td>5</td>
<td>145 ms</td>
<td>112 ms</td>
<td>126 ms</td>
</tr>
<tr>
<td>6</td>
<td>127 ms</td>
<td>130 ms</td>
<td>109 ms</td>
</tr>
<tr>
<td>7</td>
<td>95 ms</td>
<td>123 ms</td>
<td>106 ms</td>
</tr>
<tr>
<td>8</td>
<td>130 ms</td>
<td>127 ms</td>
<td>110 ms</td>
</tr>
<tr>
<td>9</td>
<td>127 ms</td>
<td>120 ms</td>
<td>98 ms</td>
</tr>
<tr>
<td>10</td>
<td>134 ms</td>
<td>123 ms</td>
<td>116 ms</td>
</tr>
<tr>
<td>11</td>
<td>116 ms</td>
<td>138 ms</td>
<td>105 ms</td>
</tr>
<tr>
<td>12</td>
<td>113 ms</td>
<td>145 ms</td>
<td>103 ms</td>
</tr>
<tr>
<td>13</td>
<td>100 ms</td>
<td>107 ms</td>
<td>131 ms</td>
</tr>
<tr>
<td>14</td>
<td>123 ms</td>
<td>112 ms</td>
<td>128 ms</td>
</tr>
<tr>
<td>15</td>
<td>117 ms</td>
<td>107 ms</td>
<td>115 ms</td>
</tr>
</tbody>
</table>

The values of FPS (Table 6.4) also show how the performance brutally dropped when compared to Scenario 1. In only one of the tests the average FPS surpassed the 24 FPS, which demonstrates that the prototype does not perform very well in this scenario's situation, affecting the user experience and QoS.
Table 6.4: FPS in scenario 2

<table>
<thead>
<tr>
<th></th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>20.68</td>
<td>19.56</td>
<td>22.72</td>
</tr>
<tr>
<td>Average</td>
<td>23.38</td>
<td>22.60</td>
<td>24.75</td>
</tr>
<tr>
<td>Maximum</td>
<td>60</td>
<td>60</td>
<td>60</td>
</tr>
</tbody>
</table>

**Summary:** In this scenario, where the peers are connected in a chain fashion, it was evident that the performance of the prototype drops a lot, with latency increase exponentially and the FPS not being at the pretend value of 24 FPS. If deployed to a real world situation the prototype may not deliver a good enough QoS and a good user experience, considering this situation.

In spite of this being a worst case situation, it has a probability of happening when in a real world situation, therefore, it was important to measure how well the prototype performed under this conditions.

6.4.3 Results from Scenario 3

Because Peer 3 was the one receiving a mixed stream, it was important to measure the performance of the prototype in that peer. For that reason, the measurements of the RAM, CPU, latency and FPS were done on that peer.

Observing Figure 6.11 it is possible to conclude that the increase of RAM continues to follow the same percentage increase, in comparison to the idle state, with an increase of 58 percent for the minimum and maximum value. The values of used RAM in this scenario varied between 258 and 327 megabytes when the three tests are considered.

![Figure 6.11: RAM used by Firefox in scenario 3](image)

Once again, this is a normal increase, with an adequate browser consumption of RAM, proving that the prototype is compatible to use integrated with a web browser.

The values of the percentage of CPU (Figure 6.12) are also similar to the values of other tests,
varying between 13 and 28 percent, for the minimum and maximum usage of CPU, considering the three tests. Still, the values are in a normal interval, corroborating what has been happening in the other test scenarios.

![CPU Usage graph](image)

**Figure 6.12:** CPU used by Firefox in scenario 3

Table 6.5 shows the latency values of the third scenario. All values are lower, by a large margin, than the reference value of 150 milliseconds. This demonstrates that both the user experience and the QoS can be considered good for the prototype.

<table>
<thead>
<tr>
<th>Minutes</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21 ms</td>
<td>27 ms</td>
<td>19 ms</td>
</tr>
<tr>
<td>2</td>
<td>24 ms</td>
<td>26 ms</td>
<td>25 ms</td>
</tr>
<tr>
<td>3</td>
<td>26 ms</td>
<td>20 ms</td>
<td>25 ms</td>
</tr>
<tr>
<td>4</td>
<td>24 ms</td>
<td>21 ms</td>
<td>22 ms</td>
</tr>
<tr>
<td>5</td>
<td>19 ms</td>
<td>21 ms</td>
<td>28 ms</td>
</tr>
<tr>
<td>6</td>
<td>27 ms</td>
<td>22 ms</td>
<td>24 ms</td>
</tr>
<tr>
<td>7</td>
<td>25 ms</td>
<td>24 ms</td>
<td>27 ms</td>
</tr>
<tr>
<td>8</td>
<td>21 ms</td>
<td>18 ms</td>
<td>25 ms</td>
</tr>
<tr>
<td>9</td>
<td>20 ms</td>
<td>21 ms</td>
<td>28 ms</td>
</tr>
<tr>
<td>10</td>
<td>21 ms</td>
<td>27 ms</td>
<td>27 ms</td>
</tr>
<tr>
<td>11</td>
<td>24 ms</td>
<td>22 ms</td>
<td>19 ms</td>
</tr>
<tr>
<td>12</td>
<td>25 ms</td>
<td>20 ms</td>
<td>20 ms</td>
</tr>
<tr>
<td>13</td>
<td>21 ms</td>
<td>27 ms</td>
<td>28 ms</td>
</tr>
<tr>
<td>14</td>
<td>26 ms</td>
<td>28 ms</td>
<td>26 ms</td>
</tr>
<tr>
<td>15</td>
<td>21 ms</td>
<td>26 ms</td>
<td>23 ms</td>
</tr>
</tbody>
</table>

Table 6.5: Latency in scenario 3

Relatively to the FPS of the video (Table 6.6), there were no drops below the reference value of 24 FPS, meaning that the user had a good experience and QoS.
Table 6.6: FPS in scenario 3

<table>
<thead>
<tr>
<th></th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>24.40</td>
<td>25.96</td>
<td>25.61</td>
</tr>
<tr>
<td>Average</td>
<td>45.39</td>
<td>43.89</td>
<td>48.76</td>
</tr>
<tr>
<td>Maximum</td>
<td>60</td>
<td>60</td>
<td>60</td>
</tr>
</tbody>
</table>

Summary: The third test scenario measures how the prototype behaves when there is a mix of streams, this is, when there is a case of real-time collaboration, with Peer 2 modifying the stream that receives from Peer 1, and sending it to Peer 3.

All measurements conducted demonstrated that the prototype behaves like it was expected, delivering a good QoS and user experience, giving good indications to how it would behave in a real world scenario.

6.4.4 Results from Scenario 4

The measurements of the RAM, CPU, latency and FPS in this scenario could be done in Peer 4 or Peer 6, due to the fact of both of them being the furthest peers. In this case, it was chosen the Peer 6.

Figure 6.13 presents the variation of consumed RAM during the three tests. This variation, considering all tests, was between 267 and 329 megabytes of RAM consumed, an increase of 64 and 59 percent when compared to the idle state of Firefox.

These values of consumed RAM are similar to the ones in the other test scenarios, meaning that the prototype will not consume too much RAM.

![Figure 6.13: RAM used by Firefox in scenario 4](image)

Once again, this is a normal increase, with an adequate browser consumption of RAM, proving that the prototype is compatible to use integrated with a web browser.

The values of the graphic Figure 6.14 vary between 17 and 28 percent of CPU used, considering all
three tests. These values are similar to the ones of the other test scenarios, although there was a little increase of the minimum value it was not a significant increase.

Latency values are, once again, lower than the reference value of 150 milliseconds, as exposed in Table 6.7. This ensures a good user experience and demonstrates a good QoS by part of the prototype, even when the number of peers using the system increases.

As it can be seen in Table 6.8, only in one of the tests realized in this scenario there was an average higher than the reference value of 24 FPS. Even though the average of the other tests is not too disparate from the desired 24 FPS, this will affect the user experience as well as the QoS.

![CPU Usage](image.png)

Figure 6.14: CPU used by Firefox in scenario 4

<table>
<thead>
<tr>
<th>Minutes</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>46 ms</td>
<td>53 ms</td>
<td>37 ms</td>
</tr>
<tr>
<td>2</td>
<td>35 ms</td>
<td>43 ms</td>
<td>42 ms</td>
</tr>
<tr>
<td>3</td>
<td>45 ms</td>
<td>35 ms</td>
<td>45 ms</td>
</tr>
<tr>
<td>4</td>
<td>48 ms</td>
<td>49 ms</td>
<td>53 ms</td>
</tr>
<tr>
<td>5</td>
<td>59 ms</td>
<td>51 ms</td>
<td>35 ms</td>
</tr>
<tr>
<td>6</td>
<td>43 ms</td>
<td>45 ms</td>
<td>48 ms</td>
</tr>
<tr>
<td>7</td>
<td>43 ms</td>
<td>38 ms</td>
<td>53 ms</td>
</tr>
<tr>
<td>8</td>
<td>44 ms</td>
<td>52 ms</td>
<td>53 ms</td>
</tr>
<tr>
<td>9</td>
<td>38 ms</td>
<td>53 ms</td>
<td>44 ms</td>
</tr>
<tr>
<td>10</td>
<td>41 ms</td>
<td>48 ms</td>
<td>39 ms</td>
</tr>
<tr>
<td>11</td>
<td>40 ms</td>
<td>51 ms</td>
<td>46 ms</td>
</tr>
<tr>
<td>12</td>
<td>38 ms</td>
<td>47 ms</td>
<td>42 ms</td>
</tr>
<tr>
<td>13</td>
<td>43 ms</td>
<td>40 ms</td>
<td>38 ms</td>
</tr>
<tr>
<td>14</td>
<td>50 ms</td>
<td>52 ms</td>
<td>39 ms</td>
</tr>
<tr>
<td>15</td>
<td>48 ms</td>
<td>42 ms</td>
<td>42 ms</td>
</tr>
</tbody>
</table>
**Table 6.8: FPS in scenario 4**

<table>
<thead>
<tr>
<th></th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>18.29</td>
<td>20.18</td>
<td>20.52</td>
</tr>
<tr>
<td>Average</td>
<td>23.31</td>
<td>24.73</td>
<td>23.70</td>
</tr>
<tr>
<td>Maximum</td>
<td>60</td>
<td>60</td>
<td>60</td>
</tr>
</tbody>
</table>

**Summary:** This last test scenario also pretends to evaluate how the system would behave when there is a more significant number of peers online. In this case, with six peers, the prototype behaved at an acceptable level.

The difficulties stood out in the drop of the number of FPS when compared to other test scenarios. There was also an increase in the latency, but, considering the conditions of the scenario, that increase is normal. When deploying in a real world scenario there might be a slight decrease of performance, which may affect the overall user experience and QoS.
## Conclusion

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Summary</td>
<td>77</td>
</tr>
<tr>
<td>7.2 Achievements</td>
<td>78</td>
</tr>
<tr>
<td>7.3 System Limitations</td>
<td>78</td>
</tr>
<tr>
<td>7.4 Future Work</td>
<td>79</td>
</tr>
</tbody>
</table>
This chapter draws conclusions about the work developed in the thesis and what was achieved and demonstrated by it. Also points out the limitations of the prototype developed and what should be improved in the future to have a better and more suitable solution for the real world situations.

7.1 Summary

The objective of this work is to show that an application focused on real-time collaboration in a P2P fashion and that only needs a web browser to be used is possible to develop and a viable alternative to the common applications that use the client-server paradigm and/or need extra software to be installed.

P2P is increasingly used and it starts to become a very reliable alternative. But most of streaming applications still use the client-server architecture and/or require extra software or plugins to be installed, driving away less experienced, and patient, users and preventing P2P to grow to its full potential. By using P2P costs with web servers and bandwidth would also reduce, what would be a great advantage for companies and developers.

This is what WebRTC does, provides browser-to-browser communication, in a P2P fashion, without any plugins or extra software. It is, at time of writing, still being worked on and being standardized, but it is already possible to use it and develop applications with it. WebRTC aims to provide an open-source solution, accessible to anyone and that brings great benefits to all the parties involved.

If a peer receives the stream from various sources, much like the BitTorrent protocol, it will improve the QoS, QoE and general performance of the stream, while reducing the costs necessary with web servers and bandwidth, and with the peers needing only a web browser.

In order to prove that a system like this is in fact possible, a prototype was built. The prototype was composed of a normal HTTP server and was extended with a coordinator that acted as a connection broker and references to where the resources are located within the network. The prototype uses WebRTC, a recent technology that gives web browsers P2P capabilities.

WebRTC only starts to be used when there are peers in the network that can serve the resources needed. If not, or if the connection between peers fails, the peers will resort to the HTTP server to obtain the resources they need. For the streaming part of the prototype it is used exclusively WebRTC.

This is only possible because of the coordinator that helps peers to find other peers with the resources and to connect them to each other. As soon as a successful connection is established, the peers trade data or stream directly to each other, not involving the HTTP server.

In the scope of this work it was not implemented a prototype that contained all the characteristics defined in 4, like the PPSP or a proper authorization system or a peer receiving a stream from multiple peers. This characteristics are considered future work and should be implemented.
7.2 Achievements

With this work it was proved that it is possible to have a WebRTC system as a credible alternative to some of the more traditional systems that are currently used.

To accomplish the requirements proposed in Chapter 4 it was presented an architecture that could fulfill all of those requirements. Next it was implemented a prototype following those requirements, in order to prove the theorized architecture. It was this prototype that was used to test the viability of what was proposed.

The tests have shown that the prototype met the requirements and also that the performance is in line with what is expected of a system like this one, providing a good enough QoS and QoE, while being self-scalable.

Concluding, it was demonstrated that a system like the one of the prototype is viable and could make P2P more attractive to less experienced users because it does not require any plugins or extra software besides the web browser.

7.3 System Limitations

The prototype is not exempt of problems and some of them need to be solved. The main problem was related to the fact that a few methods and functions of PeerJS have been deprecated, making it not compatible in its entirety with the latest versions of web browsers, despite continuing to work, at time of writing, in the Firefox web browser.

Other factor that limit the use of the prototype is the bandwidth of the peers. Bandwidth present in a home is still not comparable to the one present in web servers. Due to this, a peer can not process a lot of requests, so there needs to be a system that will give more requests to peers with better bandwidth and less requests to peers with worse bandwidth. By doing this it may be possible to guarantee a good enough stream quality to every peer. The fact of peers connecting to only one other peer does not help providing the best QoS possible. There should be a mechanism for peers to connect to multiple other peers.

Although the prototype takes into account peers behind a NAT, if the NAT system is too complex there might arise some problems when connecting the peers. By using a dedicated web server for intermediate the connections or renting a TURN server this problem might be solved.
7.4 Future Work

This prototype has some issues that need to be addressed and some functionalities that can be added to make the prototype more complete and better.

The main issue is the fact of PeerJS being outdated. Because of this, the prototype already has some compatibility issues with Firefox, with some functions and methods being deprecated or soon to be deprecated. This leads to the conclusion that it is only a matter of time until PeerJS becomes fully incompatible with Firefox, urging the prototype to be ported to a more updated API.

A very important functionality to be added is for the Peer Coordinator to stop connecting peers in a random fashion and to start to take into consideration factors like proximity of the peers, Internet quality, peer load, among others. This data could be stored by the Peer Coordinator who would, when a peer requested the list, connect it to the best peer, or peers, possible. There should be also implemented the possibility of a peer receiving the stream from several other peers instead of just one peer.

Other functionality that could be added is a login system. Only users that would create an account could provide original content or alter content of other users. Also, the login system would make possible the creation of channels, users to follow other users, suggest content accordingly to the user preferences, and others. Instead of a login system, a more simple functionality is to only allow predefined peers to contribute with original content and to alter content of other peers.

More functionalities that can be implemented are the possibility of choosing other media sources instead of the webcam and the standard microphone and the capability of the peers to save the stream that they are watching on their computer, in case they want to see it again. This would allow more diverse content to be shared and for a more flexible system by giving the possibility of reviewing the content later. It should also be implemented the PPSP in the system, as it standardizes and improves the P2P streaming.
Bibliography


Listing A.1: Code to get the list of peers with the video

```javascript
(...)

peer.listAllPeersWithContent(videoOptions.hash, function(list) {
    if (list && list.length == 0) {
        getVideoFromServer(videoOptions);
    } else {
        connectAndRequest(videoOptions, list);
    }
    (...)
});
```

Listing A.2: Code for connecting and requesting data

```javascript
(...)

conn = peer.connect(chosenOne);
```
Listing A.3: Code for answering calls

```javascript
peer.on('call', function(call) {
  if (typeof remoteStream == 'undefined' && typeof localStream != 'undefined') {
    call.answer(window.localStream);
    inCallsLocal.push(call);
  } else if (typeof remoteStream != 'undefined' && typeof localStream == 'undefined') {
    call.answer(window.remoteStream);
    inCallsRemote.push(call);
  } else if (typeof remoteStream != 'undefined' && typeof localStream != 'undefined') {
    var newStream = new MediaStream(remoteStream);
    var newStreamVideo = newStream.getVideoTracks()[0];
    var oldStream = new MediaStream(localStream);
    var oldStreamAudio = oldStream.getAudioTracks()[0];
    var mix = [newStreamVideo, oldStreamAudio];
    newStreamMix = new MediaStream(mix);
    call.answer(window.newStreamMix);
    inCallsLocal.push(call);
  }
});
```
Listing A.4: Code for the peer to start streaming

```javascript
function startStream() {
    var constraints = { audio: true, video: true; }
    peer.deleteSecondaryStreamer(secondaryHash);
    navigator.mediaDevices.getUsedMedia(constraints).then(function(stream) {
        window.localStream = stream;
        mainHash = btoa('mainstreamer');
        peer.announceStream(mainHash);
        peer.announceSecondaryStream(secondaryHash);
        document.getElementById('myvideo').prop('src', URL.createObjectURL(stream));
        document.getElementById('myVideoStreamHidden').style.display = "block";
        (...)
    }, function(err) { console.log(err.name + ': ' + err.message); });
}
```

Listing A.5: Code for the peer to stop streaming

```javascript
function stopOwnStream(inCallsLocal) {
    mainHash = btoa('mainstreamer');
    secondaryHash = btoa(peer.id);
    peer.deleteStreamer(mainHash);
    peer.deleteSecondaryStreamer(secondaryHash);
    document.getElementById('myVideoStreamHidden').style.display = "none";
    (...)
    localStream.stop();
    for (var i = 0; i < inCallsLocal.length; i++) {
        inCallsLocal[i].close();
    }
    var inCallsLocal = [];
    if (btoa(document.getElementById('calltoid').value) != '' && remoteStream != 'undefined') {
        secondaryHash = btoa(document.getElementById('calltoid').value);
        peer.announceSecondaryStream(secondaryHash);
    }
}
function showTheirStream () {

    if (secondaryHash != btoa(document.getElementById('calltoid').value) &&
        typeof secondaryHash != 'undefined' && secondaryHash != btoa(peer.id)) {

        peer.deleteSecondaryStreamer(secondaryHash);
    }

    secondaryHash = btoa(document.getElementById('calltoid').value);

    var randomPeer = [];

    peer.listAllSecondaryStreamers(secondaryHash, function(list) {

        randomPeer = list[Math.floor(Math.random() * list.length)];
    });

    navigator.mediaDevices.getUserMedia(constraints).then(function(NULL) {

    var call = peer.call(randomPeer, NULL);

    call.on('stream', function(stream) {

        $('#theirvideo').prop('src', URL.createObjectURL(stream));

        window.remoteStream = stream;
    });

    window.existingCall = call;
    outCalls.push(call);

    if (typeof localStream == 'undefined') {

        peer.announceSecondaryStream(secondaryHash);
    }

}).catch(function(err) { console.log(err.name + ': ' + err.message); });
};
Listing A.7: Code to stop watching a stream

```javascript
function stopTheirStream(outCalls, inCallsRemote) {
    secondaryHash = btoa(document.getElementById('calltoid').value);
    peer.deleteSecondaryStreamer(secondaryHash);

    (...)

    for (var i = 0; i < outCalls.length; i++) {
        outCalls[i].close();
    }

    for (var i = 0; i < inCallsRemote.length; i++) {
        inCallsRemote[i].close();
    }

    remoteStream.stop();

    var outCalls = [];
    var inCallsRemote = [];
}
```

Listing A.8: Code to see the peers that are streaming

```javascript
$(function() {
    $("#streamslist").click(function() {
        $("#streamlist ").empty();
        hash = btoa('mainstreamer');
        var streamList = "";

        peer.listAllStreamers(hash, function(list) {
            for (i = 0; i < list.length; i++) {
                streamList += "<li >" + list[i] + "</li>";
            }

            $("#streamlist").append(streamList);
        });
    });
});
```