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

Emanuel Sérgio Ruivo Almirante

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 grew exponentially, being responsible for the majority of the traffic on the World Wide Web (WWW). 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 and evaluates 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 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.

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

1 Introduction

One of the most important aspects of the Internet is the possibility to collaborate with anyone in the network. 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 system which means that there is a single point of failure and the server becomes a bottleneck, causing problems of scalability and, sometimes, of QoS.

A solution to mitigate this problem is by using P2P, which, theoretically, works better as more users are online. 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 the main server. Even so, P2P systems have some disadvantages, like, for example, needing extra software or plugins to be installed, which may cause compatibility problems between different devices.

Recently a new technology named WebRTC has emerged. WebRTC 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. WebRTC has already been adopted by the majority of web browsers, like Google Chrome, Mozilla Firefox, and Opera, immediately delivering WebRTC to a huge user base, and making it attractive to developers.

Using both the P2P to provide real-time collaboration, and the WebRTC to simplify the use of P2P for everyone and enable it to be used in any device, at any time, this system would allow real-time collaboration using live streams, and would present itself as a self-scalable solution by decreasing the high costs with bandwidth, because each user would also act as a server. Additionally, the combined use of these technologies provides the necessary

- *Emanuel Sérgio Ruivo Almirante, nr. 70569,*
  E-mail: emanuel.almirante@ist.utl.pt,
  Instituto Superior Técnico, Universidade de Lisboa.

June 2017

tools to develop applications that are easy to integrate with the ones already being used, making it simple to transition to a P2P system.

2 Related Work

A few relevant projects have been developed in this field. The most important will be described in this section.

2.1 Integration of WebRTC and Session Initiation Protocol (SIP)

The work proposed by Segeč *et al.* [1] shows 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. The architecture of the solution is shown in Figure 1.

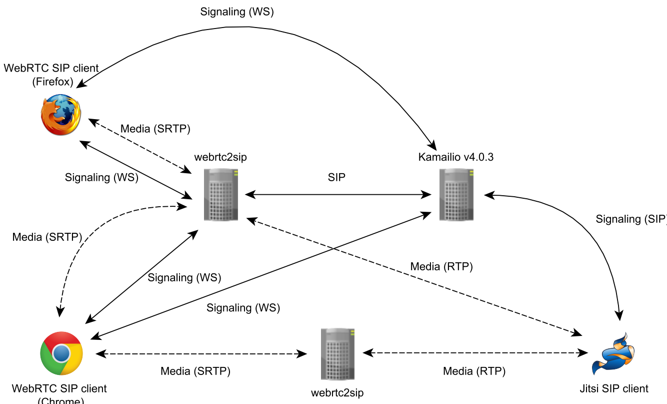

Figure 1. Architecture of WebRTC with SIP [1]

The WebRTC client signaling functionalities are implemented using WebSocket SIP Application Programming Interface (API). The WebRTC client uses a WebRTC SIP application created by the authors and it has WebSocket and SIP characteristics for signaling. The signaling server used is a Kamailio SIP proxy server, that supports the WebSocket protocol.

It is also necessary to have a media gateway to work as a translating mechanism to translate the different protocols supported by Firefox (supports Datagram Transport Layer Security-Secure Real-Time Transport Protocol (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 this case, it will be used a *webrtc2sip* gateway.

In the experimental tests, 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.

2.2 WebRTC Technology Overview and Signaling Solution Design and Implementation

Sredojev *et al.* [2] implemented a videoconferencing and chat application that was able to connect peers through the web browsers without any external plugins, using WebRTC.

The overall architecture of the project is very simple, as can be observed in Figure 2. It is only necessary a signaling server, two peers and a way for the peers to communicate (signal) with the server.

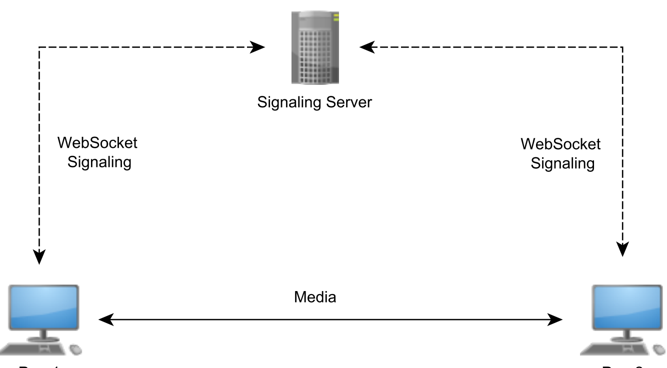

Figure 2. Overall architecture of WebRTC [2]

The Signaling Server is a WebSocket server, written in Node.js. The application for the peers was implemented using the WebRTC API. For communicating with the Signaling Server, it is used the WebSocket protocol.

Before a connection between two peers is established, it is exchanged local and remote descriptions of the audio and video media information. For this there will be used the *setLocalDescription()* method and the *setRemoteDescription()* method, that are in the API.

When the exchange of Session Description Protocol (SDP) and Interactive Connectivity Establishment (ICE) messages is over, the connec-

tion is established and the peers can communicate without using the Server as an intermediary.

In the experimental tests, the application worked perfectly and can be concluded that the implementation of the project was a success.

2.3 P2P Live Video Streaming in WebRTC

The project described in [3] was developed to see if it would be possible to implement live video streaming into web applications, running on a browser, using WebRTC. Figure 3 illustrates the architecture of this project.

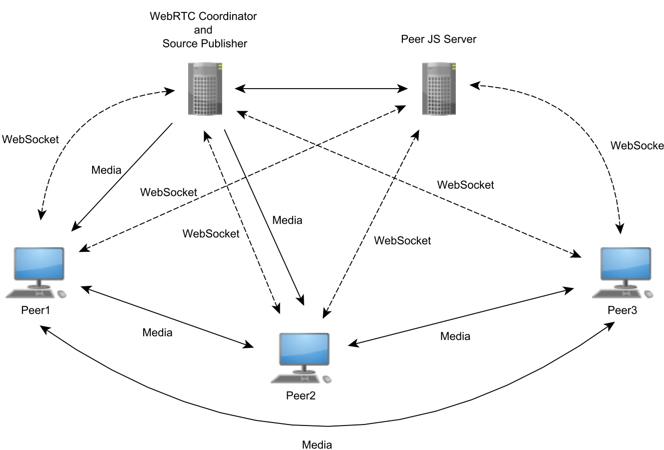

Figure 3. Overall architecture of [3]

The WebRTC Coordinator maintains the WebRTC network and, in this project, the WebRTC Coordinator is the source publisher too. The PeerJS Server is a complement to the WebRTC Coordinator, supporting it in the establishment of connections between peers.

A user that wants to join the peer network registers at the PeerJS Server and at the WebRTC Coordinator, using an ID generated by itself. After, the WebRTC Coordinator will select two peers in the WebRTC network and tells the peer trying to connect who they are. The peer opens a "RTCDataChannel" to the peers and connects. If there are no peers, the peer will subscribe to the source publisher.

The tests showed a big discrepancy between the total packets and the combined valor of control packets received, redundant packets received and delivered packets received, which can be explained by the fact that it was used User Datagram Protocol (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.

3 System Architecture

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. A representation of the proposed architecture can be observed in Figure 4.

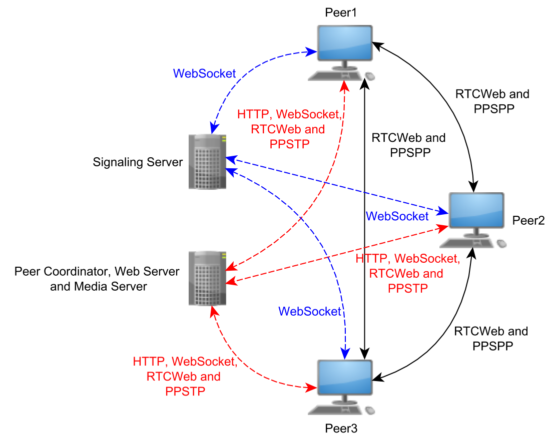

Figure 4. Architecture for the proposed solution

3.1 Server Components

This server will be composed by three modules with distinct functions and functionalities: the Peer Coordinator, the Web Server, and the Media Server.

Peer Coordinator:

In a simple way, the Peer Coordinator will be the responsible for registering the peers, to give the authorization of peers to collaborate with a video stream, to keep an update list of streamers and sources.

The registration of a peer is processed when it reaches the website, being assigned an ID by the Peer Coordinator. As soon a peer registers with the Peer Coordinator three tables are created.

One table has all the connecting information needed, like the peers ID and its Internet Protocol (IP) address. The second table maps the existing files that can be shared between the peers with the actual peers that have them. The third table helps remove peers from the network. It maps all the peers present in the network with the resources they have, in this case, all the content they are source of.

The Peer Coordinator also works as an intermediary for passing the SDP from a peer to another peer, due to the fact of the IP address of peers not being passed from the Peer Coordinator to the peers, for security purposes. 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 sources of those streams.

Web Server:
It is a simple server, with no special characteristics, that distributes HyperText Markup Language (HTML) and it directs the peers to the Peer Coordinator, for them to be registered. Also provides a Graphical User Interface (GUI) to the peers for them to interact with the system.

Media Server:
The Media Server will serve the content through a normal Hypertext Transfer Protocol (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. Peers will only get the content from it if there are no peers with the content.

3.2 Peer Components

Two modules compose the peer: 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.

Connection API:
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 Session Traversal Utilities for Network Address Translation (STUN) server to use. In case a STUN server can not resolve the address of a peer, there will be the transmission of an error message to the peer.

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. 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 arriving. The module will reassemble it and display it to the peer.

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 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 5 seconds, in this solution.

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 Network Address Translation (NAT) and firewalls, so that, no matter where a peer is, it can always connect to another peers.

3.4 Web Application

There needs to be a web application that will allow peers to take full advantage of the system. This web application will be very simple, leaving space to be improved and adapted as necessary.

4 System Implementation

This section details the implementation of the components described in the previous section. It should be noted that the Signaling Server will not be described because in this implementation it is used a public server provided by Google.

It was used PeerJS, that wraps the implementation of WebRTC on the browser in an easy-to-use and configurable P2P connection API; Node.js, an open-source, cross-platform runtime environment for developing server-side applications and to build scalable network applications; and Express.js, a web application framework for Node.js that provides a robust set of features for web and mobile applications.

4.1 Server Components

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.

Eight extra functionalities must also be added to the connection broker, for the requirements established for Peer Coordinator to be met:

- 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.

The “Announce” functionalities require no response interpretation. The “Get” functionalities will return a list with IDs of the peers that have the content or a list with IDs of the peers that are streaming.

The Media Server is nothing more than a normal HTTP server that sends a prerecorded video to the clients that access the website, if there are no peers with the content on the network.

Listing 1. HTTP web server for public files

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

Listing 2. Code for the web server

```
var server = app.listen(8000, function
    () {
    var host = server.address().
        address;
    var port = server.address().port;
});
```

The Web Server will be in the same node as the Media Server. In Listing 2 it is explicit the code for the Web Server, to be able to listen to requests from the various peers.

4.2 Peer Components

The Connection API sends an HTTP request to the `/peerjs`, to handle 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 functions shown in Listings 3 and 4 to be able to implement the extra functionalities.

Listing 3. Listing all peers with content

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

Listing 4. Listing all peer streamers

```
peer.listAllSecondaryStreamers(
    secondaryhash, 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 collaboration part of the system is affected.

The first action of the Content Loader is to register the peer with the Peer Coordinator obtaining a random ID.

Listing 5. Registering with the Coordinator

```
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 had to be replaced by a *data-src* attribute.

To request the video to a peer it is generated the video's hash, based on its Uniform Resource Locator (URL), and then the request is sent to the Peer Coordinator.

Listing 6. Video selector attribute

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

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.

The Content Loader is also the responsible for all the logic behind the P2P collaboration part, being the responsible for announcing that a peer is streaming, that is the source of a stream, to request the list of peers streaming, all the other requirements necessary.

4.3 Web Application

The web application will allow the peers to interact with the system. It is a very simple interface, implemented using HTML, Cascading Style Sheet (CSS) and jQuery.

In Figure 5 is shown the main page of the web application, with the initial video being the main element of the website.

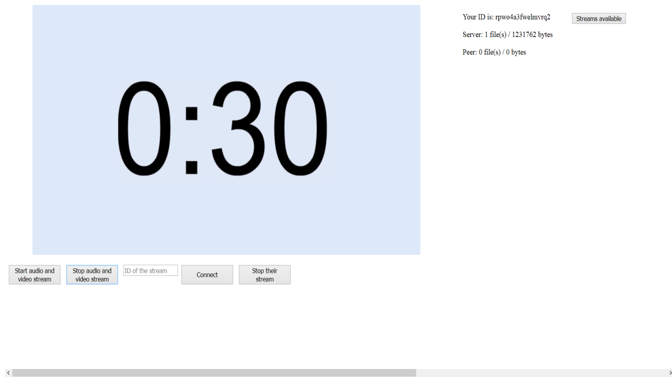

Figure 5. Interface of the main page of the web application

In Figure 6 is possible to observe how the interface will look after a peer becomes a streamer.

When watching a stream, that stream will appear in the place of the video shown in Figure 5.

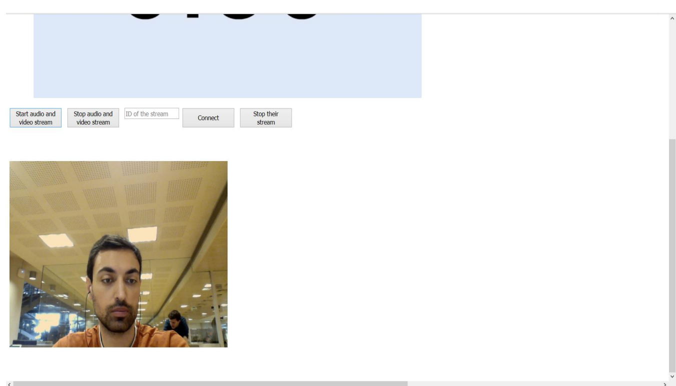

Figure 6. Interface of the web application when streaming

5 Experimental Evaluation

The tests performed have the objective of measuring the Random Access Memory (RAM) and Central Processing Unit (CPU) usage, the latency of both video and audio, and the Frames Per Second (FPS) of the video.

The prototype was deployed in a virtual machine with 256 megabytes of RAM and running Ubuntu Operating System (OS) and Node.js as the prototype's server. Peers are each virtual machines with have 512 megabytes of RAM and will run Ubuntu OS, with a GUI.

5.1 Reference Values

To have values of reference for comparison, it was tested how much RAM and CPU Firefox utilizes when it is in an idle state.

The RAM values fluctuated between 163 megabytes and 207 megabytes. The CPU values fluctuated between 0 percent and 3 percent. The latency, according to [4], should not surpass the 150 milliseconds, and the minimum FPS acceptable are 24 FPS, according to [5].

5.2 Scenario 1

Peer 1 starts by registering and download the video file from the Server. Then 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 visualize the stream of Peer 1 for 15 minutes.

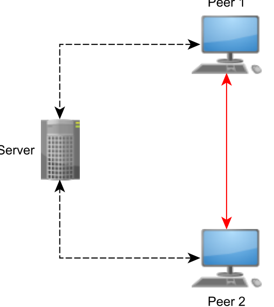

Figure 7. Diagram of the first scenario

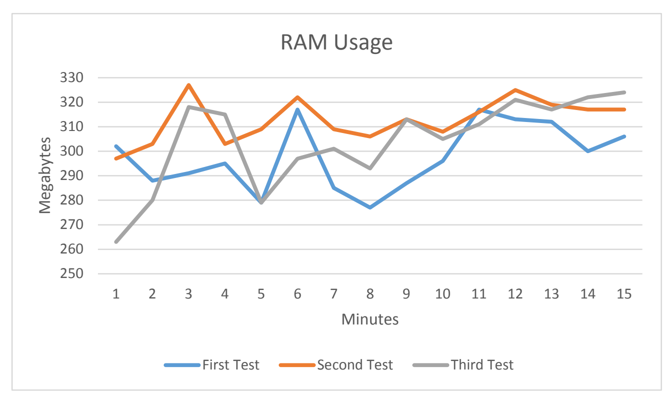

Figure 8. RAM used by Firefox in scenario 1

RAM values had an increase of around 58 percent in comparison to the idle state (Figure 8). It can be considered a normal increase.

CPU values had an astronomical increase compared to the idle state, although it is nothing out of the ordinary (Figure 9).

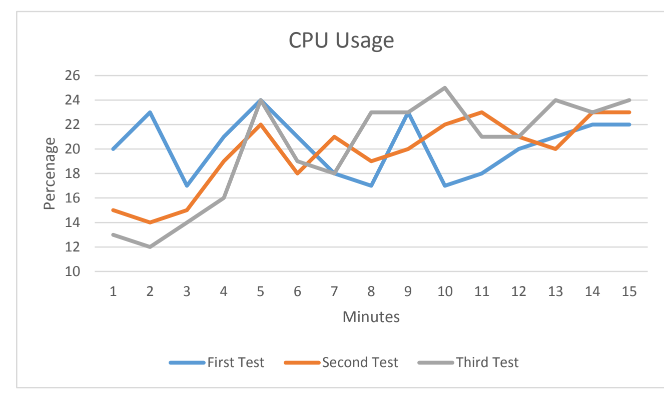

Figure 9. CPU used by Firefox in scenario 1

Concerning latency, the minimum value obtained was of 12 milliseconds and the maximum 22 milliseconds. The average FPS was

of 42.35 FPS. These values comply with the reference values.

5.3 Scenario 2

Peer 1 will register with the Server and download the video file. After 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. After, they will start to visualize the stream of Peer 1, connecting to each other in a sequentially way, like it can be observed in Figure 10.

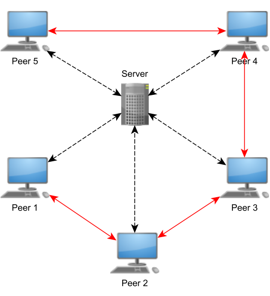

Figure 10. Diagram of the second scenario

RAM values had an increase of around 33 and 60 percent, respectively, from the minimum and maximum values in comparison to the idle state (Figure 11). It can be considered a normal increase.

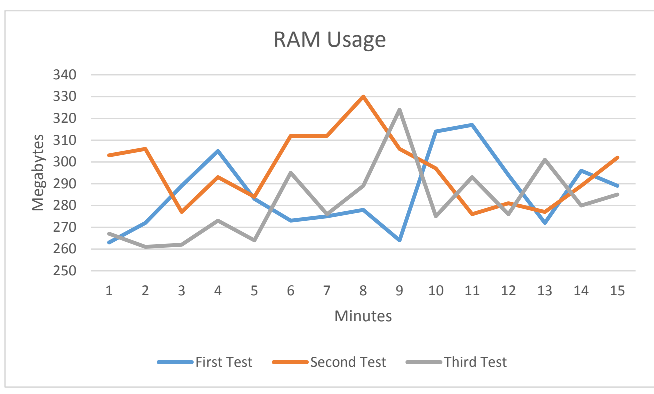

Figure 11. RAM used by Firefox in scenario 2

CPU values varied between 17 and 28 percent, values similar to the ones in Scenario 1, which may be consider normal in these circumstances (Figure 12).

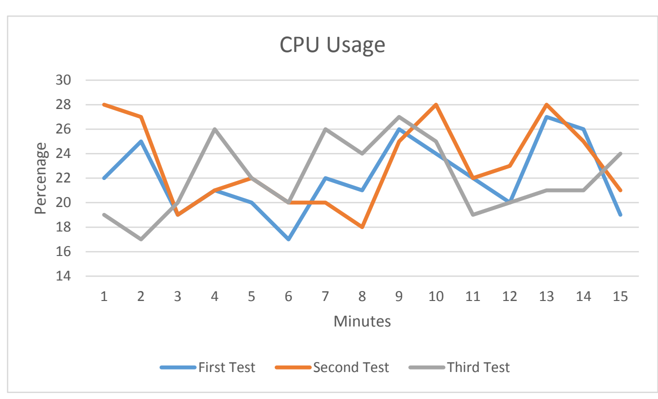

Figure 12. CPU used by Firefox in scenario 2

Concerning latency, the minimum value obtained was of 95 milliseconds and the maximum 145 milliseconds. The average FPS was of 22.60 FPS. FPS values do not comply with reference values, but are very close, which means that the QoS may still be good enough.

5.4 Scenario 3

Peer 1 will register with the Server and download the video file. After 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 from of Peer 2 itself, and will be sent to Peer 3.

Peer 3 will then enter the swarm, register with the Server and download the video file from one of the peers, and will start to visualize the stream of Peer 2 for 15 minutes.

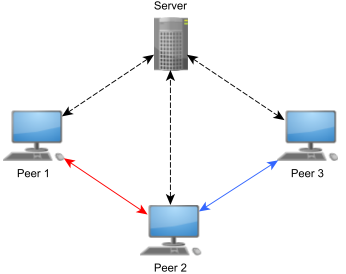

Figure 13. Diagram of the third scenario

RAM values had an increase of around 58 percent in comparison to the idle state. It can be considered a normal increase.

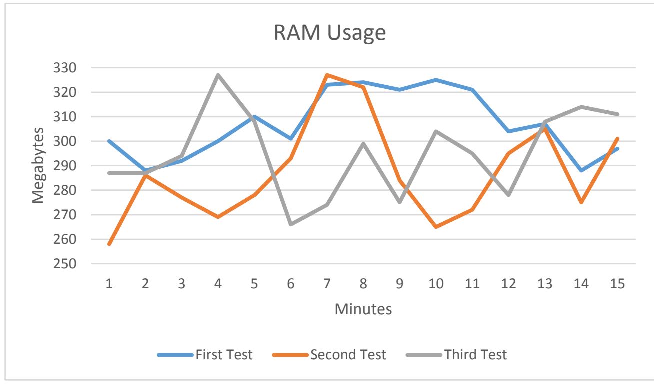

Figure 14. RAM used by Firefox in scenario 3

The values of the percentage of CPU are also similar to the values of other tests, varying between 13 and 28 percent, which can be considered normal normal values (Figure 15).

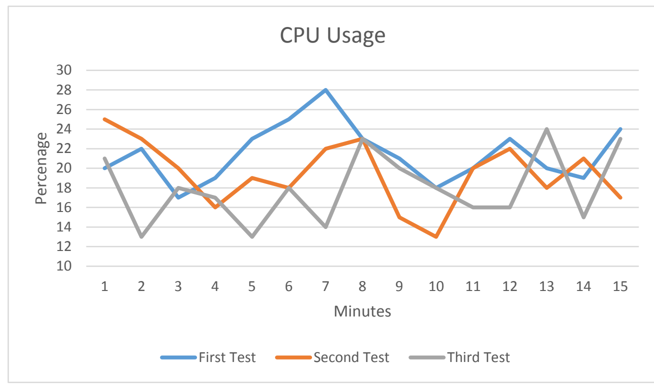

Figure 15. CPU used by Firefox in scenario 3

Concerning latency, the minimum value obtained was of 18 milliseconds and the maximum 28 milliseconds. The average FPS was of 43.89 FPS. These values comply with the reference values.

5.5 Scenario 4

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

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. This stream will be a mix of the video stream that Peer 2 receives from Peer 1 and the audio stream from of Peer 2 itself, and 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.

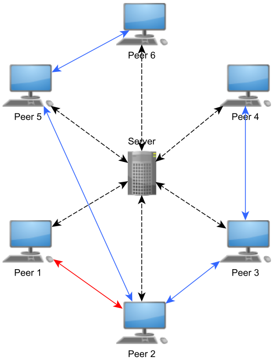

Figure 16. Diagram of the fourth scenario

RAM values had an increase of around 64 and 59 percent in comparison to the idle state (Figure 17). It can be considered a normal increase.

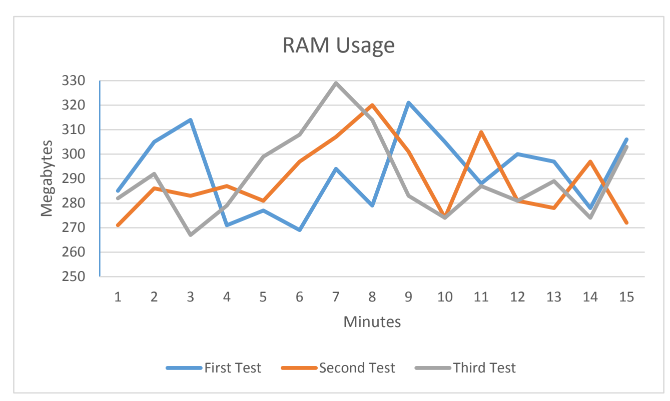

Figure 17. RAM used by Firefox in scenario 4

The values of the graphic Figure 18 vary between 17 and 28 percent of CPU used, which are very similar to the other test scenarios and can be considered normal.

Concerning latency, the minimum value obtained was of 35 milliseconds and the maximum 53 milliseconds. The average FPS was of

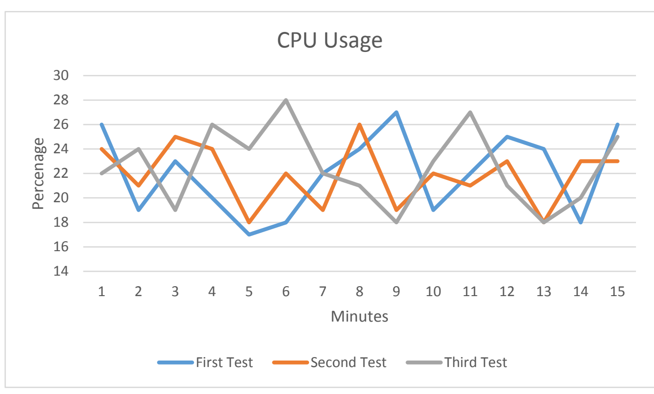

Figure 18. CPU used by Firefox in scenario 4

23.31 FPS. FPS values do not comply with reference values, but are very close, which means that the QoS may still be good enough.

6 Conclusion

WebRTC is an open-source solution that can provide browser-to-browser communication, in a P2P fashion, without any plugins or extra software, accessible to anyone and that brings great benefits to all the parties involved.

In order to demonstrate of a P2P streaming solution based on WebRTC a prototype was developed and tested in this work. The prototype consists of a normal HTTP server extended with a "coordinator" that acted as a connection broker for where the resources are located within the network.

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. But as soon as a successful connection is established between peers, they trade data or stream directly to each other, not involving the HTTP server.

6.1 Future Work

The main issue found during the development was related to PeerJS being outdated, with some functions and methods deprecated, or soon to be deprecated, urging the prototype to be ported to a more updated API. The Peer Coordinator should also start to take into consideration factors like proximity of the peers, Internet quality, peer load, among others, instead of connecting peers in a random fashion.

A login system could also be implemented, making only registered users to be able to provide original content or alter content. Instead, a more simple functionality is to only allow predefined peers to contribute with original content.

The possibility of choosing other media sources can also be implemented, as well as additional the Peer-to-Peer Streaming Protocol (PPSP) functionalities, because it standardizes and improves the P2P streaming.

Acknowledgments

I want to start by thanking 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 and colleagues for all the support and friendship since the beginning of my journey in Instituto Superior Técnico.

Lastly, I wish to thank profoundly to all my family, especially to my grandparents, and to my parents for the support during the good and the bad moments.

References

[1] P. Segeč, P. Palúch, J. Papán, and M. Kubina, "The Integration of WebRTC and SIP: Way of Enhancing Real-Time, Interactive Multimedia Communication," in *Emerging eLearning Technologies and Applications (ICETA), 2014 IEEE 12th International Conference on*, 2014, pp. 437–442.

[2] B. Sredojev, D. Samardzija, and D. Posarac, "WebRTC Technology Overview and Signaling Solution Design and Implementation," in *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on*, 2015, pp. 1006–1009.

[3] F. Rhinow, P. P. Veloso, C. Puyelo, S. Barrett, and E. O. Nuallain, "P2P Live Video Streaming in WebRTC," in *Computer Applications and Information Systems (WCCAIS), 2014 World Congress on*, jan 2014, pp. 1–6.

[4] Tim Szigeti and Christina Hattingh, "Quality of Service Design Overview," 2004. [Online]. Available: http://www.ciscopress.com/articles/article.asp?p=357102{&}seqNum=2

[5] P. Bakaus, "The Illusion of Motion," 2014. [Online]. Available: https://paulbakaus.com/tutorials/performance/the-illusion-of-motion/