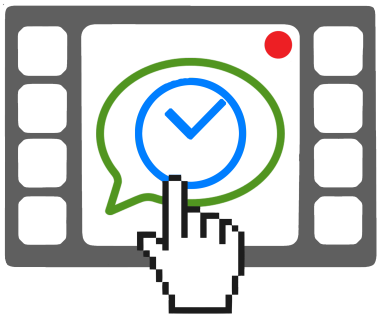
TÉCNICO LISBOA



Hyper-linked Communications: WebRTC enabled asynchronous collaboration

Henrique Duarte Lopes Rocha

Dissertation submitted to obtain the Master Degree in

Communications Networks Engineering

Supervisors: Prof. Ricardo Jorge Feliciano Lopes Pereira
Eng. Paulo Guilherme Gonçalves Chainho

Examination Committee

Chairperson: Prof. Paulo Jorge Pires Ferreira
Supervisor: Prof. Ricardo Jorge Feliciano Lopes Pereira
Member of the Committee: Prof. João Pedro de Faria Mendonça Barreto

May 2016

Acknowledgments

I would like to thank my thesis supervisor Ricardo Pereira and co-supervisor Paulo Chainho for their active support and ideas that were crucial for the development of this thesis.

I should not forget all the people involved on the prototype's test phase, they not only tested but also gave valuable feedback that we used to improve the quality of our prototype.

Also, I would like to show my gratitude to the *Kurento* team for developing the streaming server that I've used, their impressive help support and receptiveness to discuss new functionalities.

Not less important, I should thank all IST employees that maintained the space I used during this journey.

Lastly, I should thank all the people that supported me psychologically and made me going my own way.

Thank you!

Abstract

The Hyper-linked communications concept applies much of the hypermedia concepts, widely used on Web content. This paradigm allows to synchronize, structure and navigate communication content integrated into voice and video calls.

WebRTC technology allows real time communications between web browsers without the need to install additional software. The nature of web browser applications already follows the hypermedia concept, which makes *WebRTC* the ideal technology to apply the hyper-linked communications concepts. The web browser platform provides an abstraction layer that makes it possible to create applications that run independently from the operating system. The native support for *WebRTC* in operating systems extends its usage to outside the web browser, allowing for the exploration of functionalities for which web browsers provide poor support, such as video recording and massive information storage.

Our goal was the development of an application targeted to the web platform, resorting to *WebRTC*, that leveraged the hyper-linked communications by providing a multi-party conference environment enriched with multiple media types, collaborative text editors, time annotations, instant messaging, ability to superimpose hyper-content to video and the possibility to playback communications.

In this document, we present the current State Of The Art in hyper-linked communications and related technologies, propose and implement an architecture for an hyper-linked communication application based on *WebRTC*. This work was evaluated by users, who reported that they liked to use it and thought it to be extremely innovative.

Keywords: WebRTC, asynchronous communications, collaboration tools, timeline navigation, non-linear video-conferencing, video recording

Resumo

O conceito de comunicações hiper-ligadas aplica muitos dos conceitos de hiper-media, bastante utilizados no conteúdo *Web*. Este paradigma permite sincronizar, estruturar e navegar sobre conteúdo proveniente de communicações integrado em chamadas de voz e vídeo.

A tecnologia Web Real-Time Communication (WebRTC) permite realizar comunicações em tempo real entre navegadores web sem a necessidade de instalar *sofware* adicional. A natureza das aplicações *web* já usufruem dos conceitos de hiper-media, o que faz do WebRTC a tecnologia ideal para aplicar o conceito de comunicações hiper-ligadas. A plataforma providenciada por navegadores *web* disponibiliza uma camada de abstração que torna possível correr aplicações independentemente do sistema operativo. O suporte nativo do WebRTC nos sistemas operativos extendem a sua utilização para fora do contexto dos navegadores *web*, possibilitando explorar funcionalidades que os navegadores *web* suportam de uma forma limitada como a gravação de vídeo e armazenamento de informação em massa.

O nosso objectivo neste projecto é desenvolver uma aplicação, recorrendo ao WebRTC, que utiliza o conceito de comunicações hiper-ligadas de forma a enriquecer as comunicações em conferência com vários tipos de media, editores colaborativos, anotações temporais, envio de mensagens, possilidade de sobrepor contéudo ao vídeo e voltar atrás no tempo.

Neste documento, apresentamos o actual Estado de Arte nas comunicações hiper-ligadas e as tecnologias relacionadas, propomos e implementamos uma arquitectura para uma aplicação com comunicações hiper-ligadas baseada no *WebRTC*. O nosso trabalho foi avaliado por utilizadores que, para além de terem gostado da experiência com a nossa aplicação, acharam o nosso protótipo bastante inovador.

Palavras-chave: WebRTC, comunicações asíncronas, ferramentas colaborativas, navegação temporal, vídeo-conferência não linear, gravação vídeo

Contents

List of Figures

List of Tables

List of Acronyms

AES Advanced Encryption Standard

API Application Programming Interface

BOSH Bidirectional-streams Over Synchronous HTTP

CBC Cipher Block Chaining

CPU Central Processing Unit

CRUD Create, Read, Update and Delete

CSS Cascading Style Sheets

DOM Document Object Model

DTLS Datagram Transport Layer Security

DVD Digital Versatile Discs

FMS Flash Media Server

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

ICE Interactive Connectivity Establishment

IM Instant Messenger

IoT Internet of Things

IPv4 Internet Protocol Version 4

IPv6 Internet Protocol Version 6

IP Internet Protocol

JSON JavaScript Object Notation

JVM Java virtual machine

KMS Kurento Media Server

MCU Multipoint Control Unit

MPEG Moving Picture Experts Group

MVC Model-View-Controller

NAT Network Address Translation

OT Operational transformation

P2P Peer-to-peer

PSTN Public Switched Telephone Network

RAM Random-access memory

QR Quick Response

REST Representational State Transfer

RFC Request For Comments

RTCP Real Time Control Protocol

RTSP Real Time Streaming Protocol

RTMFP Real-Time Media Flow Protocol

RTMP Real Time Messaging Protocol

RTP Real-time Transport Protocol

SAMI Synchronized Accessible Media Interchange

SCTP Stream Control Transmission Protocol

SDP Session Description Protocol

SigOfly Signaling-On-the-fly

SIMPLE SIP for Instant Messaging and Presence Leveraging Extensions

SIP Session Initiation Protocol

SMIL Synchronized Multimedia Integration Language

SMS Short Message Service

SRTCP Secure RTCP

SRTP Secure Real-time Transport Protocol

SRT SubRip Text

SSL Secure Sockets Layer

STUN Session Traversal Utilities for NAT

SVG Scalable Vector Graphics

TCP Transmission Control Protocol

TLS Transport Layer Security

TMN This Means Nothing

TURN Traversal Using Relays around NAT

UDP User Datagram Protocol

URL Uniform Resource Locator

VoIP Voice Over IP

WebRTC Web Real-Time Communication

WebVTT Web Video Text Tracks

XAML eXtensible Application Markup Language

XEP XMPP Extensions

XHTML eXtensible Hypertext Markup Language

XML eXtensible Markup Language

XMPP Extensible Messaging and Presence Protocol

Chapter 1

Introduction

1.1 Background

Since the early days of Human History, we tried to communicate over far locations, from smoke signals to letters delivered by messengers. Real-time communications were limited or even nonexistent. Despite all the efforts made to improve communications, written communication could never replace face to face communication. With the advent of the telephone network, communications have taken a very important step for us to feel more connected with whom we communicate. Still, only the human voice was not enough, and the invention of cameras and consequent video digitalization were a huge step for real time communications.

In the past, handwritten documents were limited to a writer per page at a time. Writing a book collaboratively was a difficult task due to synchronization between writers. Today, we can achieve more. It is possible to write a document collaboratively, correct spelling mistakes without wasting physical resources, restructure text at any moment, add a video to a newspaper article and more. Although much of what was said seems banal nowadays, none of this was possible before the computer's invention.

As Martin Geddes states, "No computer in our lifetimes will ever rival a human voice's capacity to conveying rich and complex social and emotional meaning" [1]. Although nothing replaces the physical contact with a person while we communicate, we are at a time when we can do more than just visual and verbal communication. Hypermedia can be added to video and voice in order to extend its value. The concept of structured voice and video synchronized with hypermedia is called hypervoice [1].

As communications technologies appeared, we adapted the way we communicate. The purpose of this project is not the replacement of the current video and audio communications, but to enrich them with hyper-media content.

With the advent of WebRTC and its successive integration with web browsers, it became possible to develop video conference web applications without plugins. This presents a range of possibilities on what can be implemented using already existing web technologies.

Furthermore, real time communication applications can make a significant difference on business, education and health sectors by providing tools for developing teaching and learning online, teamworking and socializing web applications.

1.2 Proposed Solution

A real time system is a huge source of information that requires much attention from its users. In this context, an application that provides a way to remember our past communications would be a strong tool for not only to catch what we lost but also to enhance our knowledge.

Our goal in this project is to develop an application targeted to the web platform, resorting to WebRTC, that leverages the hyper-linked communications by providing a video conference environment enriched with interactive and non-interactive discrete media types such as images, subtitles, forms and all types of content that can be added using HyperText Markup Language (HTML)5, Cascading Style Sheets (CSS)3 and *JavaScript* including continuous media types such as video, music and animations.

One of the key features of this project is the ability to navigate in time in order to reproduce the conversation again or introduce hyper-content to it such as time annotations, interactive lists of topics and subtitles. We call this non-linear video-conferencing. In this context, we also provide a simpler method for creating and synchronizing hyper-content using Quick Response (QR) codes.

In addition to this conference environment, which provides different functionalities than traditional conference environments such as *Skype* and *Google Hangouts*, we also enable a collaborative text editor and a chat that support sending time hyper-links (to navigate to a location in time) and files to conference participants.

Furthermore, another relevant feature is the possibility to compose multiple video streams into a single one, which enables adding more users to conference rooms without impacting the clients' performance. Users can change to individual streams on demand or automatically to the talking users.

1.3 Thesis Contribution

Making it clear, this project aims to complement current audio, text and video communications in order to create rich and collaborative interfaces with the ability to add more content on a future time (*e.g.* creating time annotations for improving content search) in order to increase its value. It is also important to highlight another goal of this project, which is the ability to navigate in time by rewinding communications, fast-forward and jump to certain points.

A web application with an easy to learn user interface was developed to accomplish solving our problem. Our application, unnamed yet, is targeted at web browsers that are compatible with only standard technologies like JavaScript, WebRTC, HTML5 and CSS3. Any additional plug-in was avoided, *JavaScript* libraries were preferred as they can be downloaded on-the-fly.

We have presented an architecture that can meet our goals, implemented the respective prototype and tested it with real users and performance benchmarks.

According to Martin Geddes, the quality of the interaction worsens as the number of users increase[1]. In our testing phases we quantifyed and qualifyed the impact of increasing users on the interface and performance of our prototype.

All the problems faced during the development and limitations were reported on the thesis so that a future project better then ours can be easily and better developed.

1.4 Outline

This rest of this document is structured as follows:

- **Chapter 2** describes the previous work in the field.
- **Chapter 3** describes the system requirements and the architecture for an Web Application that fulfills the goals of this thesis.
- **Chapter 4** describes the implementation of our Web Application and the technologies chosen.
- **Chapter 5** presents the evaluation tests performed and the corresponding results.
- **Chapter 6** summarizes the work developed and proposes future work.

Chapter 2

Related Work

This section is structured as follows. Section 2.1 describes the problems that real time communications face on nowadays internet, namely the Internet Protocol Version 4 (IPv4) address exhaustion and the client server model constraints. Section 2.2 describes the WebRTC technology and the protocols needed to implement our project. Section 2.3 addresses the signaling component of chat applications, which is not defined on WebRTC specifications. Section 2.4 presents the evolution of multimedia content until the hypermedia, its capabilities, synchronization mechanisms and interactivity. Section 2.5 explores streaming protocols for non-interactive multimedia and how to introduce the interactive component, another important aspects are the ability to control the time flux of a stream and collaborative application development.

2.1 Early days of the Internet and its remaining flaws

The need to build a global communications network in an age when almost nobody had access to that technology and the unpredictability of the number of future users, lead to some protocols not being suitable for the explosive growth and proliferation of users that followed. IPv4 limits the number of public addresses in such a way that today they are scarce [2]. Today we live in the Internet of Things (IoT) era. Users and companies may have multiple devices that require Internet connectivity, such as personal computers, smartphones, tablets, traffic sensors, public surveilance systems, health monitoring tools and so on. All those devices combined cannot be publicly addressed by IPv4.

To this end, one way to overcome the IPv4 address scarcity problem was the development of a mechanism that groups multiple address into a single one, the machine that is assigned that address is then responsible for redirecting messages to members of its group using their private addresses, each connection in the private network is identified publicly by the same Internet Protocol (IP) address with a different port. This technique is known as Network Address Translation (NAT) (Figure 2.1).

Initially NAT offered an alternative to address exhaustion and a minimal sensation of security. However, due to their current wide usage, NATs weaknesses are being exposing at the application layer, namely impacting applications that require direct communications between two private networks.

There are four types of NAT implementations[3]: *Full Cone NAT*, *Restricted Cone NAT*, *Port Restricted Cone*

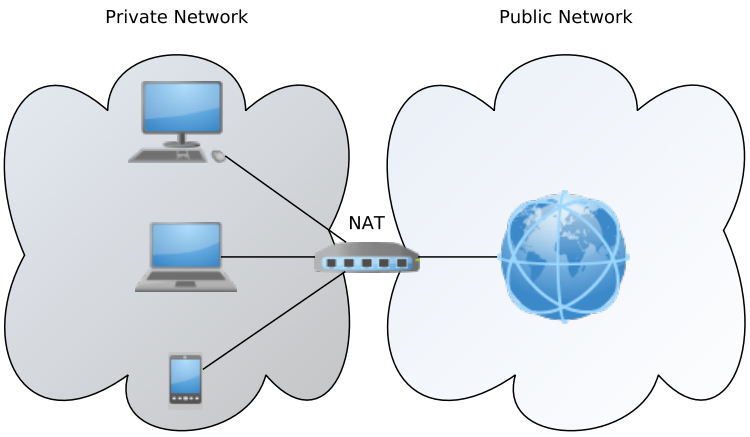


Figure 2.1: Network Address Translation

NAT, *Symmetric NAT*.

Full Cone NAT maps each public IP address and port to a private IP address and port. Any external host can communicate with private hosts through their mapped public address and port. This represents the least restrictive type of NAT and, as we will see later, the unique type of NAT that enables real time communications from point to point.

Restricted Cone NAT requires that a private client must first send a message to an external host before it can receive messages from the same host. With this type of NAT, the private client can be contacted from any port of the same external host.

Port Restricted Cone NAT works in the same way as Restricted Cone NAT, but it only allows communications from the same external host's IP address and port, ignoring all messages from other applications within the same external host.

Symmetric NAT maps different ports for each connection. As we will see later, this type of NAT represents a problem on real time communications.

Non-Symmetric NATs became the common configurations on the Internet. As a direct result, problems started to appear: the amount of ports that IP makes available is also small compared to our current needs; worse than that, NAT also difficults end-to-end communication, forcing most applications that follow this model to be implemented ineffectively.

Unless the router that performs as NAT has forwarding rules to every desired ports of each user, applications behind a NAT are prevented from receiving incoming connections from the public network, which forces them to behave as a client of a client-server model.

Applications based on multimedia and peer-to-peer file sharing have been one of the most strained by the NAT technique. It is also important to highlight that those kind of applications require real time communication in order to achieve the best performance.

Session Traversal Utilities for NAT (STUN) and Traversal Using Relays around NAT (TURN) [4] servers are a possible solution to overcome NAT. None of those can establish direct connections on multiple level NATs (Figure 2.2).

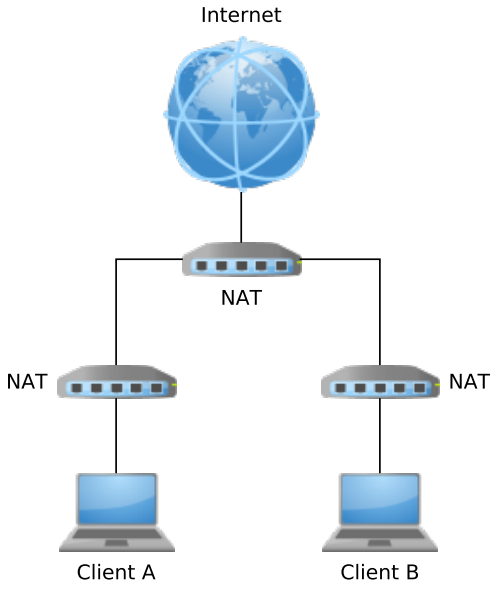


Figure 2.2: Multiple level NATs

STUN servers are quite simple. They receive requests from NATed clients, with the source address of a request being the public address that NAT mapped to the client. STUN servers will then reply to the client, providing the mapped public address, so it knows its associated public IP address and port. Symmetric NAT changes IP port for each different connection, for that reason, when the STUN servers reply with the IP address and port of their connection, it will be useless for clients to use them on other application connections. That is why Symmetric NAT represents a problem for peer-to-peer communications.

On the other hand, TURN uses public servers to relay traffic between private endpoints. It may use a Peer-to-peer (P2P) network relay to find the best peer, but after that, the behavior is much like client-server. Direct communication is only achieved by STUN when NAT is a type *full cone*. Interactive Connectivity Establishment (ICE) is a technique that uses STUN when direct communications are possible and TURN while a direct communication is not possible.

Most of client-server applications are not affected by NAT when the servers are public, but they're inadequate for real time communication between two private endpoints. Clearly TURN requires a more expensive relaying infrastructure and, in most cases, more network usage, leading to a worse quality of service. The requirements of real time video communication makes this kind of model unsuitable.

When connection is established, either in a direct or indirect way (via TURN servers), WebRTC came to simplify how audio and video are transmitted through web browsers.

2.2 Real time communications

WebRTC is an open source technology that defines a collection of standard protocols and *JavaScript* Application Programming Interface (API)s for web browser based real time communications without installing any additional application or plug-in. Table 2.1 shows the protocols that WebRTC rely on.

Some operating systems such as *Android*, *iOS*, *Linux*, *OSX* and *Windows* implement native WebRTC libraries, extending the usage of WebRTC to applications outside the web browser. This native support can help to implement applications that record video and audio streams for further playback.

WebRTC defines three main APIs: MediaStream, PeerConnection and DataChannel.

- **MediaStream** allows the browser to access the camera, microphone and the device's screen.
- **PeerConnection** acquires connection data and negotiates with peers.
- **DataChannel** provides a channel for exchanging arbitrary data with other peers.

WebRTC uses User Datagram Protocol (UDP) for transporting data, which provides lower latencies than Transmission Control Protocol (TCP), but is not reliable and does not assure packet order and integrity. Stream Control Transmission Protocol (SCTP) and Secure Real-time Transport Protocol (SRTP) are used for streaming data, providing a mechanism for congestion control and partial reliable delivery over UDP. All transferred audio, data and video must be encrypted with Datagram Transport Layer Security (DTLS) symmetric keys. DTLS provides the same security guarantees as Transport Layer Security (TLS).

TLS does not support independent packet decryption[5], for that it requires a reliable transport channel, typically TCP. The decryption of a packet depends on the previous packet, which for unreliable transport protocols like UDP may represent a problem, either due to packet loss or different reception order.

DTLS is similar to TLS, but is used on top of UDP. The main difference is the inclusion of a sequence number per packet that is used for packet re-ordering on reception and protects from duplicated packets. If a packet sequence number is less than the expected sequence number the packet is discarded. If a packet sequence number is greater than the expected sequence number the packet may be enqueued or discarded. By knowing the sequence of messages that are sent and received in DTLS, timers are used for packet retransmission avoiding acknowledgment messages.

Table 2.2 shows an overview of the transport protocols. WebRTC's *DataChannel* is built on top of SCTP, which is encapsulated by DTLS. DTLS encapsulation provides confidentiality, authentication and integrity to the transfered data. A *Data Channel* has one incoming stream and one outgoing stream, providing bidirectional communication. Each data channel direction can be configured for reliable or unreliable transmission, the same

Table 2.1: WebRTC protocol Stack

<table>
<tr><td colspan="3"></td><td colspan="2">MediaStream</td><td>DataChannel</td></tr>
<tr><td>XHR</td><td>SSE</td><td>WebSocket</td><td colspan="2" rowspan="2">SRTP</td><td>SCTP</td></tr>
<tr><td colspan="3">HTTP 1.X/2.0</td><td colspan="2">Session (DTLS)</td></tr>
<tr><td colspan="3">Session (TLS) - optional</td><td colspan="3">ICE, STUN, TURN</td></tr>
<tr><td colspan="3">Transport (TCP)</td><td colspan="3">Transport (UDP)</td></tr>
<tr><td colspan="6">Network (IP)</td></tr>
</table>

can be done for order delivery and priority. which can also be defined for improving the quality of service of a particular stream over the others.

WebRTC's *MediaStream* is built on top of SRTP, which requires an external mechanism for key exchange. DTLS keys are negotiated on handshake in order to achieve a secure connection. The new keys derived from DTLS handshake are seized for SRTP encryption, the remaining SRTP communications are done through UDP without using DTLS.

WebRTC aims to provide a standard platform for real time audio and video on the Web. It arrives at a time when several proprietary products are well established. *Skype*[1] is an application that allows video, voice, instant messaging and multi-party communication over proprietary protocols, its main strength are the amount of users that are using it nowadays and the ability to perform voice calls to the Public Switched Telephone Network (PSTN). But compared to *Skype*, WebRTC applications do not need to be pre-installed.

Google Hangouts[2] is another popular video multi-party conference web application. In the past, in order to use *hangouts* on a web browser a plug-in had to be installed. Nowadays hangouts is using WebRTC. *Google Hangouts* supports viewing videos on *youtube* synchronously, drawing collaboratively, creating music and playing multi-player games. These applications are implemented with *Adobe Flash*.

Jitsi Meet[3] is a WebRTC collaborative application that uses *Jitsi Videobridge* for high quality and scalable video conferences and supports shared document editing. *Jitsi Meet* allows a great amount of users in the same conversation by identifying the current most active participant users and, by consequence, reducing the video and audio quality for all the other users. *Jitsi Videobridge* is a server that enables multi-party video calls.

2.3 Signaling: meet and get to know

Signaling is the process by which applications exchange connection information about peers and servers, their capabilities and meta-data. In particular, WebRTC does not implement signaling, as different applications may require different protocols and there is no single answer that fits all problems. As a consequence, multiple options are available for filling the missing WebRTC's signaling component, which can be performed using Session Initiation

[1] `http://www.skype.com/`(accessed June 1, 2015).
[2] `http://plus.google.com/hangouts`(accessed June 1, 2015).
[3] `http://jitsi.org/Projects/JitsiMeet`(accessed June 1, 2015).

Table 2.2: Overview of transport protocols

	TCP	**UDP**	**SCTP**
Reliability	reliable	unreliable	configurable
Delivery	ordered	unordered	configurable
Transmission	byte-oriented	message-oriented	message-oriented
Flow control	yes	no	yes
Congestion control	yes	no	yes

Protocol (SIP), Extensible Messaging and Presence Protocol (XMPP), *WebSockets*, *Socket.io*[4] or by implementing a custom protocol.

2.3.1 WebRTC

WebRTC uses Session Description Protocol (SDP) [6] to define peer connection properties such as types of supported media, *codecs*, protocols used and network information. An SDP offer describes to other peers the expected type of communication and its details, such as used transport protocols, codecs, security and other.

One of WebRTC signaling's requisites is bi-directional communication. Hypertext Transfer Protocol (HTTP) uses a request-response paradigm, where a request is sent by the client, followed by a server response. Sometimes it is required that some information be obtained in real time, but we saw, some NAT's do not support callbacks from servers, preventing them from notifying clients as soon as an event occures. One technique to overcome this problem is polling. Polling consists on sending periodic messages to which the server responds immediately with empty content or fresh information. Text and presence messages are unpredictable, if the time between periodic requests is short, most of the time the server will return empty results wasting network bandwidth and energy. On the other hand, if the time between periodic requests is large, newer messages may arrive too late.

A technique called long polling consists on making the server hold the request until there is fresh information or expiring it after some time. As soon as it receives the reply, the client makes another request. Long polling technique results on a better network usage and a faster server response, but both simple polling and long polling requests are sent with HTTP headers, which add data overhead and can be noticed especially for short sized messages.

The *WebSocket* protocol allows bi-directional communications over a full-duplex socket channel [7], by other words it supports sending and receiving data simultaneously. *WebSocket* handshake phase specifies a HTTP header in order to upgrade to *WebSocket* type of communication, but the remainder messages are exchanged without HTTP headers, which leads to much smaller messages and better network usage. WebSockets may not be available on every web browser, frameworks like *socket.io* and *SockJS*[5] fall back to using HTTP long polling when there is no support for *WebSockets*.

Bidirectional-streams Over Synchronous HTTP (BOSH) [8] is a technique based on long polling that uses two socket connections and allows sending client messages to the server while a previous request is held. The BOSH specification assumes that a connection manager is implemented to handle HTTP connections. This connection manager is basically a translator from HTTP to raw message so that the server may be implemented as if this communication is performed over TCP. When the connection manager holds for a response for too long, it responds with an empty body, this technique prevents an HTTP session from expiring when the client is waiting for a response, thus expanding the session time. Expiring sessions can be expensive due to the overhead of establishing new connections, which is even worse when HTTP is used over Secure Sockets Layer (SSL).

If the server is holding a request, it maintains a second connection to receive more requests from the same client. The request on hold returns immediately with a possible empty body leaving its socket free, while the second connection serves the polling loop. The exchange of roles of those two connections allow to pull data from

[4] `http://socket.io/`(accessed June 1, 2015).
[5] `http://github.com/sockjs`(accessed June 1, 2015).

multiple contexts instead of being locked in just one.

2.3.2 Session Initiation Protocol

SIP [9] is a protocol used for negotiation, creation, modification and finalization of communication sessions between users. SIP follows a client/server architecture with HTTP like messages and it can be used as a signaling protocol. The advantage of SIP is the ability to make video and voice calls between the telephone network and applications over IP networks.

The working group SIP for Instant Messaging and Presence Leveraging Extensions (SIMPLE)[6] proposed the creation of SIP extensions, namely presence information [10] and instant messaging [11].

SIP is used in Voice Over IP (VoIP) applications due to its compatibility with the PSTN. Service providers are making their SIP infrastructures available through WebSockets[7]. Frameworks like *jsSIP*[8], *QoffeeSIP*[9] and *sipML5*[10] are used on the client side to parse and encode SIP messages, making SIP accessible to web based applications. SIP with *WebSockets* can be used as a signaling method for WebRTC applications, it allows web browsers to have audio, video and Short Message Service (SMS) capabilities like mobile phones. For instance, it is possible to inter-operate web communications with SIP networks, mobile and fixed phones.

2.3.3 Extensible Messaging and Presence Protocol

XMPP was initially developed for instant messaging and presence (Jabber[11]). It is nowadays an open technology for standardized, decentralized, secure and extensible real time communications. XMPP messages are eXtensible Markup Language (XML) based, which is attractive for applications that need structured messages and rich hypermedia. Another advantage of XMPP is the addition of extensions, for example *XEP-0096* [12], which adds file transfer capabilities between two entities, and *XEP-0045* [13], which enables multi-user chat. XMPP's bi-directional communication over HTTP is achieved through BOSH [14]. This kind of communication is also possible through WebSockets [15]. Today, multiple XMPP server implementations exists, such as: *ejabberd*[12], *Metronome*[13], *Openfire*[14] and *Prosody*[15]. *Ejabberd* is the server that implements more Request For Comments (RFC) specifications and XMPP Extensions (XEP)s[16].

One of the advantages that XMPP based applications offer is the possibility to send messages across different servers. An XMPP user is identified by its *Jabber ID*, which is defined by the pair username and server name in the form *username@server*.

The user profile is represented in terms of Data Forms [16] protocol and managed (registration and update) through *Info/Query* (IQ) [17] requests to the XMPP server, which responds with an IQ response containing information to retrieve the registration fields needed to fill its profile (eg username, password, telephone).

[6] `https://datatracker.ietf.org/wg/simple/documents/`(accessed June 1, 2015).
[7] `https://tools.ietf.org/html/draft-ietf-sipcore-sip-websocket-10` (accessed March 25, 2016)
[8] `http://jssip.net/`(accessed June 1, 2015).
[9] `http://qoffeesip.quobis.com/`(accessed June 1, 2015).
[10] `http://sipml5.org/`(accessed June 1, 2015).
[11] `http://jabber.org/`(accessed June 1, 2015).
[12] `http://jabberd.im/`(accessed June 1, 2015).
[13] `http://lightwitch.org/metronome`(accessed June 1, 2015).
[14] `http://igniterealtime.org/projects/openfire/`(accessed June 1, 2015).
[15] `http://prosody.im/`(accessed June 1, 2015).
[16] `http://en.wikipedia.org/wiki/Comparison_of_XMPP_server_software`(accessed June 1, 2015).

An important aspect of this model is the ability to define a user as a set of attributes that can change dynamically in order to fit the Instant Messenger (IM) application's needs.

The connection to the XMPP server can be done through the same web server as the user is connected, but this web server would handle a lot of connections and that would have a great impact on the overall system performance. On the other hand, users can directly connect directly to XMPP servers from web applications using the *JavaScript* library *strophe.js*. [17]

Typically a web application consists in multiple web pages which are navigated by users. Each time a web page is accessed, either from a new context or a transition from a previous page, its context is cleared except for local storage and cookies. The *JavaScript* context is cleared, including the XMPP connections performed by *strophe.js*. As such, an automatic mechanism is required for avoiding the implicit user reauthentication. One solution for reauthentication can be achieved by storing the user's JabberID and password on local storage and every time a page is accessed the authentication is performed without the users knowledge. Clearly this solution represents security flaws as the local storage can be easily accessed locally or by performing a cross site scripting attack to reveal all the local storage. The same problem arises if the credentials are stored on cookies.

An alternative solution is known as Session Attachment [18], which requires that a *session identifier*(SID) together with a *initial request identifier (RID)* be passed to *strophe.js* in order to re-connect to the same stream on XMPP server. Either SID and RID are unpredictable and, particularly, *RID* changes on every request making it worthless if a user maintains more than one tab opened, for example for multiple conversations at the same time.

Another alternative would be the development of our solution in a single web page, which would increase drastically the complexity of our web application.

We know that an XMPP based application could simplify our work just by using functionalities that XMPP already implement, but we do not need all of them. In fact, we just need a subset of XMPP features, namely a simple way to register users, access and edit user's information, create chat rooms, send messages and access to presence information.

2.3.4 Signaling-On-the-fly

Another interesting approach for signaling would be Signaling-On-the-fly (SigOfly) which allows inter-domain real time communications, while abstracting the protocol used [18]. SigOfly provides inter-domain communication by making use of the *Identity Providers* of each peer. The caller entity downloads a page with all the code needed, also known as messaging stub, to communicate with the called party. This code contains an implementation of the signaling protocol used in order to communicate to the called peer. If the called party domain is being overused, it is possible to switch the caller and called parties role, after that the called entity downloads the stub code from the caller domain instead. SigOfly is an approach very flexible because participants on a video call are not tied to just one type of signaling implementation. Another important aspect of SigOfly is the ability to perform multi-party conversations either through a *Mesh Topology* or a *Multipoint Control Unit*.

[17] `http://strophe.im/strophejs/` (accessed May 6, 2016).
[18] `https://metajack.im/2008/10/03/getting-attached-to-strophe/` (accessed May 6, 2016).

2.4 Hypermedia: more than words, more than images

Since the early days of video technology, one of the problems raised consisted on how to add more information onto video without generating multiple versions. This section examines technologies that allows different ways to present multimedia content in such a way that it change based on synchronization amongst other multimedia elements or user interaction.

Hypertext is a type of text that provides links to texts or other types of content, these links are known by *hyperlinks*. *Hypermedia* is an evolution of *hypertext*, it includes audio, images, text and video.

Hypermedia concept brings the possibility to organize and overlay multimedia elements into a nonlinear linear structure.

In the beginning of analog video technology, the navigation over it was quite limited to simple operations such as play, stop, rewind and fast-forward. As video started to be digitalized, new operations over video emerged, such as random jumps and chapter navigation through interactive menus present on Digital Versatile Discs (DVD).

Some Moving Picture Experts Group (MPEG) implementations, like [19], added hypermedia information to empty space present on MPEG frames in order to provide interactive television, modifying the MPEG encoder and decoder in order to handle hypermedia content. Hypermedia is a concept that holds the promise of future technology and features but it is also already present in our daily lives.

Subtitles are an example of information that might be required. The need to translate movies, raised the problem whether it is appropriate to change the original video or audio. For example, subtitles should be an entity independent from the video, in order to be personalized or replaced easily.

Synchronized Accessible Media Interchange (SAMI), and SubRip Text (SRT) are two of the multiple formats for subtitles commonly supported by video players. Although those formats have styling available, they are quite limited to text.

Hyper-video is a kind of video that contains links to any kind of hypermedia, including links to skip part of it. An example of hypermedia application could be a search engine over hypermedia content, like subtitles, in order to jump to a specific time in a video or audio track. *HyperCafe* [20] was an experimental project to expose hyper-video concepts that consisted of an interactive film that enabled switching between different conversations taking place inside a cafe.

Detail-on-demand is a subset of hyper-video that allow us to obtain additional information about something that appears along the video, like obtaining information about a painting that appears in a particular segment. *Hyper-Hitchcock* [21] is an editor and player of detail-on-demand video.

In order to navigate through a dynamic video, we must be aware of time synchronization and the multiple time flows, it ils important that all time, causality and behavior rules are well defined.

HyVAL [22] is an XML based language that was proposed for modeling composition, synchronization and interaction of hypermedia. HyVAL defines defines video structure, internal video and external media objects. *HyVAL* uses a primary video stream, around which all other elements are organized and synchronized. *HyVAL*'s video structure object defines a structure derived from traditional video, which divides video into segments, scenes, shots and frames hierarchically. This approach is quite restrictive if we want to apply hyper-video concepts to videos that do not follow this structure. External media objects are linked by primary video, those objects can

represent other videos, images, text, animation and sound.

Synchronized Multimedia Integration Language (SMIL) [23] was introduced to describe temporal behavior of multimedia content, in particular, it could be used to overlay subtitles on films. With SMIL it is possible to synchronize multiple videos, either in parallel or in sequence, reproduce a different audio track, overlay user interface elements with hyper-links, among multiple other features.

SMIL is an XML based language that defines twelve modules: *Animation*, *Content Control*, *Layout*, *Linking*, *Media Objects*, *SmilText*, *Meta Information*, *Structure*, *Timing*, *Time Manipulations*, *State* and *Transitions*.

- The **Animation** module contains elements and attributes that define a time based mechanism for composing the effects of animations. For example, this module can perform changes on XML or CSS attributes like color and dimensions.
- The **Content Control** module contains elements and attributes that provide optimized alternatives for content delivery. For example, it could be used to change audio language in function of user's nationality, for videos with multiple audio channels.
- The **Layout** module contains elements and attributes for coloring and positioning media content. Other layout mechanisms are also possible, such as CSS.
- The **Linking** module contains elements and attributes for navigational *hyperlinking*. Navigation can be triggered by events or user interaction.
- The **Media Object** module contains elements and attributes for referencing rendering behavior of external multimedia or control objects.
- The **SmilText** module contains elements and attributes that define and control timed text. For example, this module could be used to create labels and captions.
- The **Meta Information** module contains elements and attributes that allows describing the SMIL document. For example, this module could be used to define movie details such as category, director, writers and cast.
- The **Structure** module defines the basic elements and attributes for structuring SMIL content. This module defines a *head* element that contains non temporal behavior information defined by *Meta Information*, *Layout* and *Content Control* modules. This module also defines the *body* element, where all temporal related module information is contained.
- The **Timing** module is the most important module on SMIL specification. Due to its complexity, it is divided into seventeen sub-modules for coordination and synchronization of media over time. The three main elements are *seq*, *excl* and *par*, that, respectively, play child elements in sequence, one at a time and all at the same time.
- The **Time Manipulations** module adds time behavior attributes to SMIL elements, such as speed, rate or time.
- The **State** module defines attributes that define the state of SMIL elements, such as element visibility, current element time, amount of repeated loops, playing state and many others.

- The **Transitions** module defines attributes and elements for transitions across multiple SMIL elements according to the *Timing* module.

The Document Object Model (DOM) is a standard API that allows easy management of documents that are organized in a tree structure, by providing Create, Read, Update and Delete (CRUD) operations over its elements and their attributes. DOM makes it easy to inter-operate between imperative and declarative programming languages [24].

Like DOM, SMIL DOM is an API for SMIL documents. Allowing CRUD operations over SMIL documents is an important feature for extending SMIL capabilities, for example for creating non-linear animations and triggering external events like *JavaScript* functions.

SMIL's modules can be used to synchronize and animate eXtensible Hypertext Markup Language (XHTML) and Scalable Vector Graphics (SVG) elements.

SMIL fits our goals for creating a multimedia rich hyper-call, but it lacks on browser compatibility. Ambulant [25] was one of the SMIL players that were developed for browsers, although this player implements most of SMIL 3.0 [26] specifications, it needs to be installed on browsers as a plug-in.

SmillingWeb [27] attempts to implement a cross platform multimedia player designed for SMIL 3.0 presentations with *JavaScript* and *jQuery* which, unlike [25], does not require a plug-in to be installed and should not have incompatibility issues. SmillingWeb already takes advantage of HTML5 and CSS3. It takes into account unsupported web browsers through the use of *Modernizr*[19], a simple *JavaScript* library that may require plug-ins if new features are not supported. But SmillingWeb just implements a subset of SMIL 3.0 and their scheduler engine loads the SMIL file only once, which could raise problems when dealing with SMIL changes due to real time communications. Another problem with SmillingWeb is pre-loading and playing elements at the correct interval of time, which is not always possible due to high latency networks leading to pauses during playback.

An alternative to SMIL's *Layout* module is to use HTML which is a markup language based on XML that is used for creating web pages. HTML alone is a very poor language when we are focused on visual appealing and interactive web pages. Languages like CSS and *JavaScript* are typically used along with HTML for improving the interaction and appearance of a web page.

CSS's goal is to separate the structure of an XML document from its appearance. CSS defines styles for XML tags based on their name, class, identifier or position. Besides static styling CSS also supports animation and transitions leading to more dynamic content.

JavaScript is an imperative object-oriented language based on *ECMAScript*. It is used mainly on client-side and executed by a web browser. *JavaScript* has its own implementation of DOM and one of its advantages is the ability to download and execute code on-the-fly without the need of pre-installed plug-ins.

JavaScript has compatibility issues among the different web browsers, leading to different behaviors. To solve that problem, there are libraries written in *JavasSript*, namely *jQuery*, that implements the same functionality for multiple browsers, masking most of the incompatibility issues.

With the emergence of HTML5, tags like *video*, *audio* and *track* allow us to play video with multiple *codecs*, audio and subtitles in Web Video Text Tracks (WebVTT) format. Another important tag is *canvas* that allows drawing graphics with *JavaScript* on a rectangle within a web page.

[19] `http://modernizr.com/`(accessed June 2, 2015).

For example, with APIs like WebGL[20], it is now possible to manipulate a three dimensional environment in the context of a hyper-call. Another example would be a collaborative spreadsheet using WebRTC. With this, hyper-calls are not limited to only audio, image, text and video, but also interaction with complex graphical user interfaces that changes over time.

SVG is an XML based format that incorporates the animation module of SMIL. Currently, SVG allows adding movement and animating attributes of elements. When embedded on HTML5, it allows dynamic changes to inner content in real time through the DOM API. Besides that, it also allows calling *JavaScript* functions on events such as animation end, mouse over and mouse click.

Back in 1995, *flash*[21] was developed for web-based animations. Introducing video support in 2002, *flash* started to grow after that. Competitor's players, at that time were focused on playing video and audio, while *flash* had vector graphics and focused on streaming *on-demand* video across multiple platforms. *VP6* was their choice of video *codec*, providing half the video size (in Byte) for the same quality and providing video quality adjusted to *Internet* connection latency. In 2010, *Adobe Flash* was the most widely used applications for reproducing live broadcast and recorded video [28], it supports progressive video download using HTTP and streaming using Real Time Messaging Protocol (RTMP).

RTMP is a TCP based protocol used for streaming audio, video and data between a Flash Media Server (FMS) and *flash* players. A bidirectional connection is established between the two in order to allow real time communications. A *flash* player can stream a webcam video to a FMS using RTMP or it can request a video stream to FMS that can either be a pre-recorded stream, live stream or data. Multiple FMS servers can be used in parallel to increase capacity and handle more streams simultaneously.

FMS can stream video and audio to one or more subscribers by sending a separate copy for each subscriber. With Real-Time Media Flow Protocol (RTMFP) it is possible to stream video directly between *flash* players, allowing a publisher to break up a stream into pieces that can be cooperatively distributed in a P2P mesh. RTMFP uses UDP to speed packet delivery, which although it is not reliable, is well suited for video streaming. Like WebRTC, *flash* players also need to apply techniques like STUN and TURN for NAT traversal.

Although HTML5, *JavaScript*, CSS and WebRTC implement some of *flash*'s features, it does not mean that *flash* will be replaced soon. Instead, both technologies can be used to develop rich *Internet* applications. It is also important to note that HTML5 is better supported in mobile devices than *Adobe Flash*.

Like *Flash*, *Microsoft Silverlight*[22] is a cross browser plug-in and platform that is used to develop rich *Internet* applications. It supports vector graphics, animation and video. Compared to *flash*, which uses *ActionScript*, *Silverlight* applications can use languages like C#, *VisualBasic* and eXtensible Application Markup Language (XAML). *Silverlight* uses a technique called *Smooth Streaming* from *IIS Media Service* that consists on delivering video in real time with adjusted quality in function of bandwidth variations and Central Processing Unit (CPU) usage.

Using technologies that relies only on web standards, like CSS, HTML5, *JavaScript* and SVG, will make possible to develop an application that solves our problem with the advantage of being compatible with a greater amount of web browsers.

[20] `http://khronos.org/webgl/`(accessed June 2, 2015).
[21] `http://www.adobe.com/products/flash.html`(accessed June 2, 2015).
[22] `http://www.microsoft.com/silverlight/`(accessed June 2, 2015).

2.5 Extending collaboration tools with time manipulation

Real time collaboration applications have become a huge help on team tasks, providing a great boost on business, research and investigation velocity. Technologies like these are appearing along these days, but they were not be possible a few years ago because technology was limited or unavailable. Although today's technology is still limited on some aspects, progress is being done in order to improve the web ecosystem, by creating standards and migrating to newer technologies.

Section 2.5.1 presents the RTP protocol how it can be used to record media streams. Section 2.5.2 describes the media types and what types of media should be streamed. Section 2.5.3 addresses how an interactive media can be recorded. Section 2.5.4 presents the collaborative environment and libraries to synchronize distributed object among users.

2.5.1 Streaming and Recording

If, for instance, one wants to rewind a real time video, recordings will be needed from whom is streaming the video. Our first concern on real time collaboration applications, besides the communication itself, is the data storage and representation. Storing multimedia content is not a viable solution because most browsers recommend limiting local storage to at most five megabytes per origin[23] .

In order to provide a way to record and playback streams, additional servers will be required to process and record the large amount of data generated by audio and video streaming.

Real-time Transport Protocol (RTP)[29] is used for streaming audio and video over IP. Multimedia content is transported on the payload of RTP messages, that contain headers for payload identification. RTP is independent from its payload type, allowing it to transport any kind of encoded multimedia. A sequence number is used for sorting received packets.

Real Time Control Protocol (RTCP) is used for controlling RTP multimedia streams, it provides bandwidth statistics and control information that can be used for changing the quality of the stream in real time.

RTP allows to change its requirements and add extensions to it with profiles. One of the most used ones is the RTP profile for audio and video [30], which lists the payload encoding and compression algorithms. This profile also assigns a name to each encoding which may be used with other protocols like SDP. Another profile for RTP is defined by SRTP, which provides encryption, authentication and replay protection for RTP traffic. The analogous secure protocol for RTCP is Secure RTCP (SRTCP).

Both SRTP and SRTCP use Advanced Encryption Standard (AES) by default, which is a symmetric-key algorithm for data encryption. Each packet is encrypted using a distinct key-stream, as otherwise, using a single key-stream with AES on Cipher Block Chaining (CBC) would make it impossible to recover from packet loss.

Two key-stream generators for AES were defined: *Segmented Integer Counter Mode* and *f8-mode*. If a packet is lost, there is no impact on other packets, as the initialization vector is obtained through those key-stream generators and it is fixed for each packet.

[23] `http://www.html5rocks.com/en/tutorials/offline/quota-research/` (accessed May 13, 2016)

RTP recorders are independent of payload encoding, they do not decode RTP packets, they record packets instead, allowing to record all video and audio formats even if they're encrypted.

Even though one on one calls are common, there are occasions when several people take part of the same video call. Multi-party video calls can be achieved on WebRTC by streaming video from each participant to all the other participants. Although this works, the bigger a conference room is, the bigger is the used bandwidth to stream video to all participants within the conference room. In this scenario, a more efficient alternative to peer-to-peer is the use of a Multipoint Control Unit (MCU).

Jitsi Video Bridge [24] receives one stream from every participant on a conference, either from a *jitsi* client or a *WebRTC* application, and redirects it to all the other conference participants, reducing the amount of data that each peer sends. Although all the participants still need to download all the streams from the *Jitsi Video Bridge* server, typically download rates are much bigger than upload rates, making this solution more feasible. *Jitsi Video Bridge* uses XMPP as a signaling protocol and its *colibri* extension [31] to reserve channels for video transmission. Despite this choice for signaling protocol, *Jitsi Video Bridge* also supports SIP.

Kurento Media Server (KMS) supports trans-coding, group communications, recording, mixing, broadcasting, applying filters and performing image and sound analysis (for example it allows face and QRcode detection). KMS functionalities are exposed through *JSON-RPC* over *WebSocket*. There are three clients available: *JavaScript* client for web browsers, *Java* client for *Java EE* servers and *JavaScript* client for *Node.js* servers. KMS supports streaming over *WebRTC*, HTTP and RTP endpoints. Another important component of KMS is *Kurento Repository*, which supports recording and playing directly from *MongoDB*. That is important for providing a scalable media storage. Unlike *Jitsi Video Bridge*, KMS does not enforce a specific signaling protocol.

2.5.2 Media Types

Media Types can be distinguished by two criteria, the first one describes a media as discrete or continuous, the second one describes it as interactive or non-interactive. A discrete media is characterized by not depending from time, and continuous media as depending on it. Interactive media is characterized by its state being changed by external events such as user interactions. For example, an image is non-interactive and discrete, while a video is continuous and non-interactive. A simple collaborative editor with just text is interactive and discrete. An animation that changes in function of user behavior is interactive and continuous.

Streaming protocols like RTP and Real Time Streaming Protocol (RTSP) were designed for continuous and non-interactive media types, such as audio and video. Discrete and non-interactive media do not need to be streamed through RTP because they do not change with time. For example, if an image appears in a specific time interval, just the HTML or *JavaScript* that will reference the image must be streamed, the image itself is then transferred through HTTP.

In order to play a stream, a player must be prepared to interpret the stream content. For interactive stream the player must download an environment, decode the RTP payload to determine the state and display it to the user. Streaming interactive media like a combination of HTML, CSS and JavaScript requires more than interpreting the code: a streamed user interface may contain an internal state that is not shown on code.

[24] `http://jitsi.org/Projects/JitsiVideobridge`(accessed June 2, 2015).

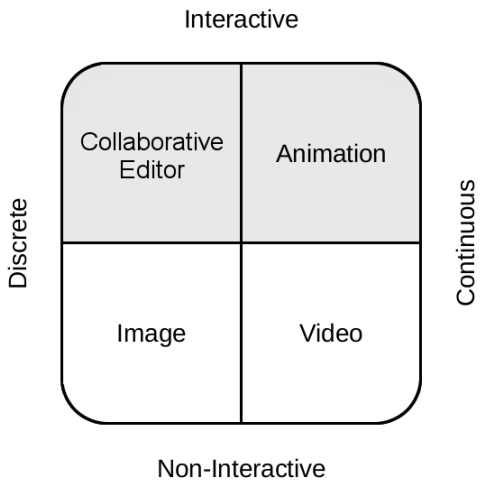


Figure 2.3: Media Types

2.5.3 Recording and Streaming Interactive Media

Mauve et al. proposed an RTP profile for real time transmission of interactive media[32]. This new profile reuses much of video and audio profile implementation, integrating the interactive component. *Hilt et al.* explained how to record interactive video with this new profile [33].

Every time an event is processed on one of the endpoints, both sender and receivers state must stay synchronized, otherwise events may behave differently. To achieve synchronization of interactive data, most packets have three types: *State*, *Delta-State* and *Event*. State packet defines the environment complete state. Delta-State packets transports just the piece of state that changed. Event packets informs that an event occurred over the interactive media.

An RTP recorder can have two operation modes, recording or playback. Traditional RTP players can do random access. In contrast, interactive RTP players must restore the environment and context at a given time. The environment is the initial state, so we can call it a non-interactive discrete media and handle it over HTTP. After the receiver has received the environment, it should calculate the state at the given time.

If the RTP recorder controls the correct data to send to the receivers, it cannot be a simple RTP recorder as it must compute the state or delta-state to send. Therefore, if the receiver receives all recorded packets, it can calculate the current state from the previous complete state. Streaming too many complete states results on more precise random accesses, but the trade-off is the higher bandwidth usage and used storage space on the recording server. On the other hand, if there are fewer complete states recorded followed by delta-states, the recorded stream will occupy less storage space, but random accesses will be less granular. It is possible to restore the media state even if messages are lost by recording and streaming the interactive media's complete state periodically.

In order to synchronize an interactive application state amongst participants, the needed objects to synchronize must be serializable and sent to other participants.

Mauve et al. concluded that the ability to extract the objects state in order to synchronize them and the ability to intercept events in order to control remote objects can be realized using the Model-View-Controller (MVC) concept[32]. This concept separates three components within an application. The *Model* represents the information

itself. The *View* component shows the *Model* to the user in a suitable and interactive way. The *Controller* represents an action from the user to the *Model*.

Using the MVC concept will it make possible to implement an interactive WebRTC application that records, play, fast forward, fast rewind, stop and jump to random positions.

2.5.4 Collaborative Environment

Google Wave was a distributed collaboration platform based on *Jupiter*[34] that adopted Operational transformation (OT) techniques. OT technology was originally developed for consistency maintenance and concurrency control over distributed objects. OT algorithms are mainly used in collaborative applications such as distributed document edition. Other Google products, such as Google Docs, are also using this type of technology. In 2010 Google stopped the development of Google Wave and released the main components as Open Source code to Apache, the project is currently known as *Apache Wave* and the reference implementation is named as *Wave in a Box*.

Among other platforms and libraries we present *ShareJS*[25], *TogetherJS*[26], *Goodow*[27], *Etherpad Lite*[28] and *otJS*[29].

ShareJS is an OT *JavaScript* library, developed by the ex *Google Wave* engineer Joseph Gentle, for collaborative text and JavaScript Object Notation (JSON) documents edition in real time. It uses *ShareDB* [30] for its backend and data model, which supports simple integration with any database. One of *ShareDB*'s integrations is over *MongoDB*[31].

TogetherJS is a JavaScript library that uses WebRTC for collaborative web applications. It uses JSON messages for OT concurrency control but it does not provide storage. *TogetherJS* uses its own servers for the signaling phase and it supports microphone and mouse sharing between users. It requires a simple server (also known as hub) that echoes messages between clients, with all the synchronization work being performed by the clients. Despite the hub's reduced complexity, *TogetherJS*'s server is implemented with *NodeJS*. If we chose a server other than *NodeJS* for our implementation, we would need to read the hub's source code and understand what changes are needed to perform on our web server in order to make it compatible with *TogetherJS*.

Goodow is a collaborative framework with its own server implementation, it supports four types of collaborative elements: String, Lists, Maps and Custom objects.

Etherpad Lite is a collaborative text editor implemented with *NodeJS* that allows not only the basic text operations but also to associate authorships to pieces of text. It supports adding functionality through plug-ins[32], for example: exporting and importing document formats such as *DOC*, *PDF*, *ODT* and *DOCX*, painting and drawing, video and audio chat using WebRTC, create sideshows, spell checking and text to speech.

otJS is a *JavaScript* library that only implements operation transformations over plain text on the client side.

[25] `http://sharejs.org/`(accessed June 2, 2015).
[26] `http://togetherjs.com/`(accessed June 2, 2015).
[27] `http://realtimeplayground.goodow.com/`(accessed June 2, 2015).
[28] `https://github.com/ether/etherpad-lite`(Accessed 20 March 2016)
[29] `http://operational-transformation.github.io`(accessed March 10, 2016)
[30] `https://github.com/share/sharedb`(accessed March 10, 2016)
[31] `https://github.com/share/sharedb-mongo`(accessed March 10, 2016)
[32] `http://static.etherpad.org/plugins.html`(Accessed 20 March 2016)

An implication of implementing just the client side is the extra effort that is necessary to implement content's persistent storage. Besides this drawback, this library is very flexible because it is not tied to a specific database or server side technology.

In summary, Table 2.3 shows a comparision of the OT libraries presented so far.

Table 2.3: Comparision between Operational Transformation libraries

Library	**Own Server**	**Own Storage**	**Operations**
ShareJS	yes	yes	text+objects
TogetherJS	yes	no	text+objects
Goodow	yes	yes	text+objects
Etherpad Lite	yes	yes	extendable
OT.js	no	no	text

2.6 Chapter Summary

In order to implement an hyper-linked communication solution, several design decisions had to be made. The limitations imposed by the *Internet's* structure, its protocols and browser's capabilities are key factors to consider when implementing a solution that allows bi-directional communications, interactive media and a collaboration environment.

Due to the use of NAT, bi-directional communications between clients have different needs from request-response based communications between clients and servers. This lead to the appearance of mechanisms such as STUN, TURN and ICE, in order to bypass the limits imposed by NAT.

WebRTC introduced an API for video, audio and data communications through web browsers without using plug-ins. However, WebRTC by itself does not define how users get to know each other nor how information flows between users. For this reason, we have studied the multiple ways we could implement this *get-to-know* mechanism which is known as signaling protocol.

With the communications establishment issue solved, we had to discuss the different types of media and what can be done with each kind in order to increase the value of communications among users. In this context, we have studied solutions and libraries that allow us to implement our prototype with time manipulation features, collaborative text edition, record and playback interactive video.

Chapter 3

Architecture

Taking into account the goals of this project and all the technology presented so far, our proposal is the development of a web application that provides communication and collaboration features in real time.

3.1 Requirements

In a general way our system's goal is to provide a multi-party video and audio conference environment that supports chat, time manipulation, collaborative text edition and hyper-content creation.

For our system, the goals include:

- **Instant Messaging** - Our application must provide a simple way to send instant text messages to the conference participants.
- **Room management** - Conference rooms can be public or private. Public chat rooms are moderated by a group of clients which initially consists only of the room creator. This type of room has no access restrictions by default, but that can be changed by its moderators. Private chat rooms will only be visible to a defined list of clients or can be accessed by clients that have a link for that conference room.
- **Stream Recording** - Ability to record and playback recorded video including all the hyper-content displayed at that time.
- **Stream Composition** - Our streaming server must support mixing multiple user streams into a single stream. The sound played on clients must be the composed sound of all participants but the video played can be switched to the individual or group view.
- **Time annotations** - Our application must give the possibility to create annotations associated to a specified time and support viewing them on a timeline.
- **Content Overlay** - Our system must allow users to superimpose hyper-content to video given a range of time. This content can be continuous or discrete and interactive or non interactive.
- **Search** - Search every objects related to a conference room such as hyper-content, time annotations and users.

- **Content Sharing** - Our application must have the ability to share files and time links among users.

- **Active talker detection** - The system must enable sound detection for showing the current speaker.

- **Collaboration** - The web application must provide a collaborative text edition tool.

- **QR Code detection** - The system must interpret QR codes in order to ease content creation.

- **NAT Traversal** - The system's services must be available to users that are inside a private network.

- **Fault tolerance** - Our system's must use a database that supports data replication.

- **Scalability** - Our system must be horizontally scalable, which means that load must be distributed across web, streaming and database servers.

Moreover we allow clients to discover chat rooms and other clients by navigating on the web pages provided by our web server. In addition users can create rooms for multi-party audio and video communication communication which is achieved by using WebRTC's *PeerConnection*.

3.2 Modules

In this section we present the several modules that were designed in order com fulfill the set requirements. Figure 3.1 presents the structure of our system which was divided into six modules:

- **Application module** - responsible for providing information about the relevant modules (*NAT Traversal* and *Signaling*) and user interface to the *Client* in the form of web pages in HTML and *JavaScript* libraries through HTTP.

- **Signaling module** - responsible for *Client* and *Stream* coordination which will be performed using *WebSockets*.

- **NAT Traversal module** - STUN and TURN are techniques used by *Client* and *Stream* modules during the *Signaling* phase which ends by establishing the connection between them.

- **Stream module** - responsible to receive and deliver multimedia content to the *Client* using WebRTC.

- **Storage module** - provides two main functionalities: store the model information and media recorded. This is the single module responsible for persistent storage. It stores user and communication data as well as all the data required among user sessions. It is also use to store all the communication streams, so that they can be viewed later.

- **Client module** - responsible for the interaction with the user.

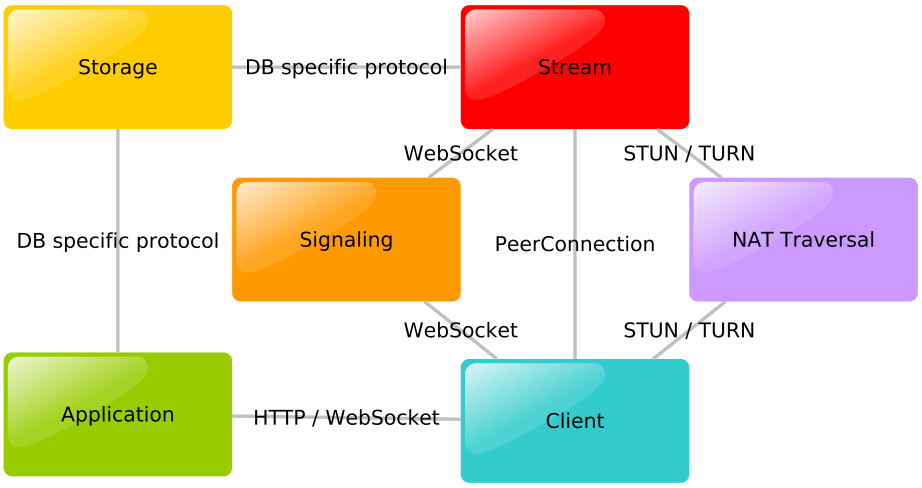


Figure 3.1: System Modules

3.3 Implementation Proposal

The infrastructure is composed by: web server, stream server, database and video repository.

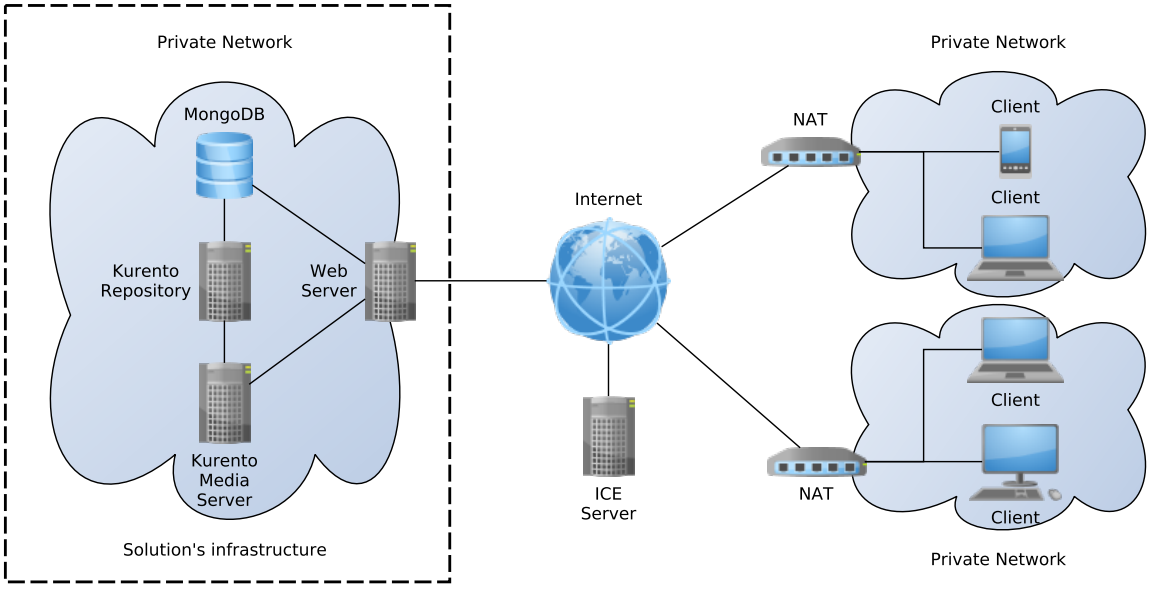


Figure 3.2: System Infrastructure

In order to simplify our solution, we propose that the *Application*, *Signaling*, *Stream* and *Storage* modules be implemented within the same server application, so it could be easy to deploy as a single virtual machine image. To this set of modules, we often call *backend*.

3.3.1 Security and authorization

Having established that the *backend* modules are placed in the same machine, that helps controlling which resources the client has permission to access as those modules are seen as a private network.

To qualify the above we provide public access to HTTP server ports, maintaining the access to other components restricted through firewall rules. In relation to the database there is no direct access from the outside. All the database information is accessed via our application server which validates the permissions of users on our system.

On the other hand, the access to our streaming servers is also restricted, but clients can connect to them after concluding the signaling phase. This signaling phase may or not proceed in function of the client's access permissions. For example, if a user is trying to access a private conference room that he is not a member of nor has an invitation link for, the signaling server refuses to start the signaling phase and the user cannot access the streaming server.

The placement of our streaming servers inside a NAT also has an important role with respect to external misuse prevention. Otherwise, placing our streaming servers could allow external clients to perform their own signaling protocol and, as a consequence, use our infrastructure without our consent.

3.3.2 Client connections

Although the delegation of processing work to clients can improve our system's scalability, we are concerned about using the least resources possible on the client side, as huge resource consumption may drain battery very fast or may even be impossible to run on mobile devices. We are aware that streaming video from clients is already a very intensive task which we cannot avoid but can improve by delegating the most intensive tasks to our servers.

In this context, each client must only have one *PeerConnection* to our streaming server and content shown to them is changed on demand either being it an individual or a composite view. This approach is centralized, as seen on Figure 3.3. Otherwise clients could follow a peer-to-peer connection, as seen on Figure 3.4, which would result on maintaining more connections and performing the composition of videos on client side.

The composition of streams on client side could be performed by receiving streams with the best quality possible from other users but, due to resizing the video of clients into a smaller region, this would result on wasting bandwidth on a quality that is not needed.

Although the peer-to-peer approach could be used on our system, we conclude that we need to record the video on our streaming server because web clients have a very limited storage and peer disconnections may result on

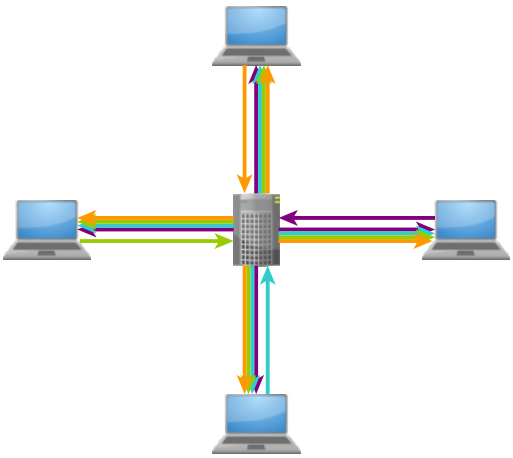


Figure 3.3: Centralized connections

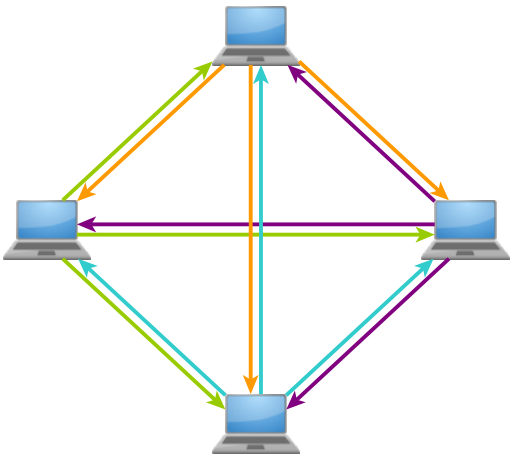


Figure 3.4: Peer-to-Peer connections

recorded video loss.

The same can be concluded to instant message delivery, each client must have only one *WebSocket* connection to the application server which consequently relays the messages to other users.

Relaying instant messages from clients through the our web servers is easier if all clients are connected to the same server because all messages can be directly delivered without sending messages across multiple web servers.

In the context of this thesis, we will not implement sending messages across web servers but, in order to allow our system to scale, we will consider that all conference participants are connected to the same server and our system is scaled by having conference rooms distributed across different servers.

3.3.3 Software choices

Furthermore, we have taken into account the compatibility between the streaming server, database and the operation transformation solutions, in order choose the appropriate framework to implement our web server.

As a result of our study about signaling protocols we have decided to exclude XMPP due to the development difficulties that it exposes when using multiple web pages, namely the user re-authentication performed each time a page is loaded and the single tab usage limitation. As a consequence we have not chosen *Jitsi Video Brigde* for stream processing as it uses XMPP.

Accordingly, by excluding *Jitsi Video Brigde* we have decided that our solution must use KMS. Our web server could be implemented easily with *NodeJS* or *Java* due to the fact KMS provides clients for both technologies. But others could also be used as KMS also exposes their API via *WebSockets*.

Due to the fact we are going to use KMS as streaming server solution, we could choose *NodeJS* or any *Java* based web framework for implementing our web application server. One of our criteria for choosing the web framework is the ability to follow the MVC paradigm which can help us to organize our code. We have decided to implement our web server with the *PlayFramework*[1] using *Java* because of our previous experience with it.

By default, *Kurento Repository* is implemented over *MongoDB*, for convenience our storage model will also use the same database.

Importantly, for the collaborative text editor, we have chosen *OT.js* due to its server and storage implementation choice independence.

For the *NAT Traversal* module, an ICE server is not required to be on our infrastructure, as a public STUN server can be used for testing our solution. Nevertheless, we recognize that for a production environment we would need to maintain our own TURN servers in order to ensure connectivity to all clients.

Not less important, on the client computers, both *Mozilla Firefox* and *Google Chrome* could be installed as web browsers. As such, both should be supported. Libraries such as *jQuery*, *Bootstrap*, *Adapter.js*, *OT.js* can be downloaded from the web server and executed on the client side using any of these two browsers.

[1] `https://www.playframework.com/`(acessed March 25, 2016)

Table 3.1: Application Architecture

<table>
<tr><td colspan="8">Application</td></tr>
<tr><td colspan="2">jQuery</td><td>HTML5</td><td>CSS3 (Bootstrap)</td><td>Signaling</td><td>ot.js</td><td></td><td>adapter.js</td></tr>
<tr><td>HTTP</td><td colspan="3">User Interface</td><td colspan="2">WebSocket</td><td colspan="2">WebRTC</td></tr>
</table>

Table 3.1 presents the application architecture and the underlying technologies seen from the user's perspective. *Adapter.js* and *jQuery* will ensure that our application is compatible with the most popular web browsers. *Bootstrap* will be used to make the user interface more appellative and responsive. With *Bootstrap* it is quite easy to develop applications that adapt to mobile devices with different screen sizes.

With respect to displaying content, the synchronization between multimedia elements will be performed through chains of *JavasSript* events or by specifying the interval of time which time content must be visible. Other animations can be implemented with SVG embedded on HTML.

3.4 Chapter Summary

In this chapter we have described the modules needed in order to implement our solution.

Several technologies were taken into account for implementing each module. We have studied the pros and cons of each technology and decided the architecture for our solution.

Chapter 4

Implementation

In this chapter we are going to present our implementation choices, the difficulties that we have faced and the strategies available to overcome them.

To this end, we are going to present and explain our database model, how we have improved it and how this model can help us reach our goals.

Subsequently, we are going to describe our signaling protocol which will make it possible to use the WebRTC's functionalities in order to implement our system's streaming features.

With the model and signaling protocol defined we are prepared to show the multiple approaches we have followed in order to implement stream recording, their drawbacks and our choices.

Afterwards we are going to describe how we show hyper-content to users, namely the multiple ways to create hyper-content and the algorithm behind the user interface synchronization, the security flaws of our content displaying mechanism and a possible solution to overcome vulnerabilities.

Not less important we also are going to describe how we have implemented our application timeline and the functionalities above it such as time manipulation and annotations management. Among other different functionalities we also are going to describe how we perform stream composition, synchronization of our collaborative text editor and implementation of our instant messenger.

Lastly, we are going to describe the deployment of our solution in respect to the hardware that we have used and the software installed onto it.

4.1 Data Model

The data model is a critical component of our solution, as a badly designed model can imply serious difficulties when implementing new features that are not part of the plans. During the course of this project, we had to redesign the model more than once in order to support new features.

In order to offer all the functionalities that we promise, some information about objects must be persistent such as users, groups,relations among users, group memberships, messages, hyper-content, recordings and collaborative editor state.

4.1.1 Schema representation

MongoDB has a slightly different terminology from relational databases. The first big difference is instead of having tables *MongoDB* stores its objects on collections. The analogous data structure to the table row is a document.

Each *MongoDB* document is represented by a *JSON* object and, as a result, each document may have different attributes within a collection. Needless to say that, by following this approach we do not need to create the collections with a well predefined schema. In fact, we do not need to define it at all but, for reasons of coherence and organization, we represent our database collections as they would have a predefined schema by following the same document structure as we will present in this section.

Similarly to relational databases, *MongoDB* requires a primary key (typically of type *ObjectId* and named *"_id"*) for each document, which is automatically assigned if not specified.

In order to define *foreign keys*, we just store them as *ObjectId*s if and only if the foreign keys point to documents within a unique collection, otherwise we need an additional attribute to specify which collection is the *foreign key* pointing at.

In respect to attributes nullability, *MongoDB* does not enforce a document's attribute to have a not null value, although, for sake of good functionality, we perform those constraints validation programmatically and as such we also represent them in our schema.

An example of schema representation can be seen on Table 4.1.

4.1.2 Generic model

For designing our model we have taken into account generic programming techniques. We observed that operations like searching for an object were quite repeated across different types of objects.

Our first decision for our model, in order to avoid repeated code, was the isolation of the object's attributes from themselves, so we could apply the search operation to a set of attributes independently of the object type. To this generic set of properties we call data (Table 4.2) and each object of this type has a reference to the owner, which is a unique identification number.

The owner's identification number by itself is not sufficient to identify an object, as objects from different types can have the same identification number. In order to solve this problem, when an object is created, its correspondent data must contain the owner's object type.

Whenever an attribute is created, its name, value and indentifiability must be specified. The set of objects that can read and write that attribute must also be defined. In regard to our permission mechanism, if the read or write sets are not specified we assume the attribute is readable and writable by everyone. Conversely if the read and

Table 4.1: Schema representation

Collection name	
◆ _id (primary key)	ObjectId
● Not nullable property name	Property type
○ Nullable property name	Property type
○ Reference to document	ObjectId
● Embedded document	Document
● Embedded list	List[Type]

Table 4.2: Generic data model

Data	
◆ _id	ObjectId
● owner	ObjectId
● properties	List[Attribute]
○ searchableValues	List[Text]

Attribute	
● key	Text
○ value	Object
● identifiable	Boolean
○ readPermissions	List[ObjectId]
○ writePermissions	List[ObjectId]

write sets are empty, nobody is allowed to read or write the attribute. Implicitly, if an entity can write an attribute, it can also read it.

In particular, if all attributes were searchable, it could be simple to search for attributes that could reveal sensible information about an object. For example if we consider that a user could have a health related attribute, searching by a disease would reveal which users could suffer from a certain disease. The leak of that kind of information could, for instance, change the agreement between users and health insurance companies. For this reason only the specified attributes as searchable will be taken into account when performing keyword searches.

Another important attribute specification is the owner identifiability, which tells us if the attribute identifies the object. This specification lets us create abstract authentication services. For example a user can login into our system by providing any attribute that identifies himself, *e.g.* the e-mail, but others are possible like the user name or cellphone number.

Not less important, in order to get an object's properties efficiently, we have created an index over the *owner* attribute. We have also created an index over the *searchableValues* in order to improve the keyword search performance.

In summary, with this model we can perform search and identification of any kind of objects, as we will see on the following models. In particular, the *user* and *group* models are using this generic model for storing their attributes.

4.1.3 User model

The user model is not tied to the user attributes (Table 4.3), the information maintained in this model is just used for authentication purposes. Passwords are not stored in plain text, instead we apply hashing and salting techniques [35] in order to make it harder to decode the password by an attacker. Accordingly, we use *SHA-1* and a random salt per user with 32 characters.

4.1.4 Relation model

A relation between two entities e_1 and e_2 is represented by the pair $e_1 \rightarrow e_2$ (Table 4.4), where e_1 is the source and e_2 is the target. This relation is said to be bi-directional if and only if it also exists the relation $e_2 \rightarrow e_1$.

Table 4.3: User model

User	
◆ _id	ObjectId
● hash	Text
● salt	Text

Table 4.4: Relation model

Relation	
◆ _id	ObjectId
● source	ObjectId
● target	ObjectId

A user can only interact with friends or with group members. In order to validate a friendship, both users must agree on that friendship, in other words it, there must exist a bi-directional relation between both users.

In order to improve the performance of queries over the *Relation* collection we have created indexes on the *source*, *target* and also on the pair composed by both attributes.

4.1.5 Group model

A conference room, which is the environment where users communicate among themselves, is represented persistently by a group of participating entities (in general users but our system allow other types of entities). A conference room is composed by only online users and does not need to be stored on database.

A group is composed by an *id*, *inviteToken* and a *visibility* as shown on Table 4.5. Moreover, a group can be public or private. If the group is public, then it is visible to all users that maintain a friendship with a member of this group. If the group is private, then it is only visible to its members.

The group membership is a special case of relation, where the target entity is always a group. When a group is created, a group membership is automatically assigned to its creator.

Entities that have a membership with a group can create more memberships by sharing an invite token or by specifying new group members. This invite token is used to create an invite Uniform Resource Locator (URL) that if shared with other users allow them to join the group. Invite tokens can be deleted or regenerated with a different value.

In order to improve performance of queries over *GroupMembership* collection we have created indexes on the *groupId*, *userId* and also on the pair composed by both attributes.

4.1.6 Message model

A message is composed by its content, time of creation and source and target identification numbers (Table 4.6). The message's target could reference any object, but our application is only handling messages to groups.

In order to query for recent messages for a given target, we use the oldest message's *_id*, which is sequential, in order to find messages with a newer *_id*.

In order to improve performance of queries over *Relation* collection we have created indexes on the *target* attribute and also on the pair composed by *target* and *_id* attributes for finding and sorting the messages received

Table 4.5: Group model

Group	
◆ _id	ObjectId
○ inviteToken	Text
● visibility	Text

GroupMembership	
◆ _id	ObjectId
● groupId	ObjectId
● userId	ObjectId

Table 4.6: Message model

Message	
◆ _id	ObjectId
● source	ObjectId
● target	ObjectId
● content	Text

by an entity more efficiently.

4.1.7 Hyper content model

During a group conversation, it is possible to create time annotations for making it easy to access that time either for searching or sharing with other users. A time annotation (Table 4.7) contains a title, the correspondent group identification number and the time itself.

The hyper content is used to synchronize content among users during a conversation. Table 4.7 shows that every hyper content must have a start and ending time, the correspondent group identification number and the content itself, in the form of text. Beside those attributes, in order to perform queries over the HTML contents with more precision, we have added an additional *searchableContent* attribute. It contains just the searchable content extracted from the content, by excluding the HTML tags and parameters using *Jsoup*[1].

In the point o view of a user, a time annotation is just a simple way to associate a time stamp to a topic in order to make it easier to find. On the other hand, the hyper-content is more complete than time annotations, except they are not visible on the timeline. In addition to time annotations, hyper-contents have a duration and content that can be superimposed to video.

For example the content showed in Listing 4.1 would produce the searchable content ”Click here!”.

Listing 4.1: Example of HTML content

```
<div class="subtitle">
    <a href="/">Click here!</a>
</div>
```

In order to improve the performance of searching content and calculate interval intersections, we have created indexes over the following sequences of attributes:

- {*groupId,start,end*} → for finding visible contents for a given time within a group (intersections).

[1] `http://jsoup.org/` (Accessed April 11, 2016)

Table 4.7: Hyper content model

HyperContent	
◆ _id	ObjectId
● groupId	ObjectId
● start	Date
● end	Date
● content	Text
● searchableContent	Text

TimeAnnotation	
◆ _id	ObjectId
● groupId	ObjectId
● title	Text
● time	Date

- {*groupId,start*} → for finding contents that starts after a given time within a group (for pre-loading).
- {*groupId,searchableContent*} → for searching contents by keywords within a group.

4.1.8 Collaborative Content model

Within a conversation, users can write documents collaboratively. As we can see in Table 4.8, each document has a content and a reference to the correspondent group.

In order to improve the performance of finding the group's content we have created and index over the *groupId* attribute.

4.1.9 Recording model

During a conversation, users may allow sharing their web cameras. By doing so, their video is stored in recording chunks. Each chunk, described in Table 4.9, represents an interval of time $T = [c^{start}, c^{end}[$ of the video stream. It also contains the media's URL, a reference to a group, an owner identification number and the correspondent *WebSocket* session id.

The media's URL is the location where *Kurento Repository* stores one chunk of video (including audio) which is used to playback on users demand.

In order to allow the same user to have different devices, storing just the URL with an associated user id is not enough. In this case, one more parameter is needed to differentiate the different *webSocket* sessions opened by the same user. For this reason we had to associate a random session identification number to each *webSocket*.

A set of chunks $S = [c_1, c_2, \ldots, c_n]$ is said continuous if $\forall c_i \in S, \exists c_j \in S$ where $j \neq i$ and $c_i^{start} = c_j^{end} \vee c_i^{end} = c_j^{start}$. A recording interval represents a continuous set of recording chunks.

In respect to find which chunks to play given a specified time (which is the current time in a user's perspective), we must select the chunk with an ending instant immediately after the current time, if the chunk's beginning is placed before the current time we have to play it instantaneously otherwise the chunk's beginning will occur after the current time and therefore we have to schedule the playback.

On the other hand, we use the fact that *_id*'s are sequential in order to determine adjacent chunks, for example, if we need to find the next chunk to play, we have to sort all elements by *_id* (which on *MlongoDB* is already sorted by default) and select a chunk with an *_id* immediately after the current one.

In order to improve the performance of searching recording chunks and calculate which chunks are going to be reproduced, we have created indexes over the following sequences of attributes:

- {*groupId,start,end*} → for finding all available chunks for a given time
- {*groupId,sessionId,end*} → for finding chunks that ends after a given time within a session

Table 4.8: Collaborative content model

CollaborativeContent	
◆ _id	ObjectId
● groupId	ObjectId
○ content	Text

Table 4.9: Recording model

RecordingInterval	
◆ _id	ObjectId
● groupId	ObjectId
● start	Date
● end	Date

RecordingChunk	
◆ _id	ObjectId
● groupId	ObjectId
● owner	ObjectId
● sessionId	Text
● start	Date
● end	Date
● url	Text

- {*groupId,owner,end*} → for finding chunks that ends after a given time within a user (for a different session) or group
- {*groupId,sessionId,_id*} → for finding chunks that follow a given *id* within a session
- {*groupId,owner,_id*} → for finding chunks that follow a given *id* within a user or group

Not less important, we have also created an index over the *RecordingInterval*'s *groupId* attribute for listing all intervals within a conference room.

4.2 XMPP experiments

Although we have used our own signaling protocol on our implementation, the first steps we made during the development of our prototype were using XMPP. We discovered the drawbacks of this approach in practice and described our learnings in the state of the art. We have found and fixed a bug present on *strophe.js* library that silenced an error raised by duplicated user registrations.

As a consequence of not using XMPP for the signaling protocol, we have defined our own protocol based on *WebSockets* for performing the stream between clients and KMS. From the KMS functionalities, our solution uses the recording feature, composite endpoints that mix multiple audio and video streams into a single one and QR code detection for creating hyper content.

4.3 Signaling Protocol

As we have already mentioned on our related work, WebRTC does not implement the signaling protocol, which is used to establish connections between peers. This connection may be direct using STUN or relayed by a TURN server in case of direct communications are not possible.

The signaling protocol plays a fundamental role on our solution. Taking into account the choices we have based on our related work, we have decided to implement our own signaling protocol in order to use WebRTC on our solution.

Although we have mentioned that the signaling protocol is used to establish connections between peers, on our system our media server (KMS) is a peer that receives video streams and sends to its connected clients.

In a general view, the signaling protocol is used to share connection and media properties between two peers. In order to ease the understanding of our signaling protocol, we have created a sequence diagram that is represented

on Figure 4.1[2].

Before the signaling protocol starts, the user must be authenticated. After the user provide the correct credentials, it receives an HTTP *cookie* from our server in order to be identified in the following HTTP requests.

After the web application server validates the user access, the signaling protocol allows the users to directly connect to the KMS, which is placed in a private network, and lets the application server and users negotiate media types and encoding information to use during the conversation. We considered implementing the signaling protocol using HTTP messages. However, they would transport extra information such as HTTP headers and would follow a request-response signaling mechanism which would not be the best option, as multiple ICE candidates can arrive at any time. Using a Representational State Transfer (REST) API over HTTP for signaling and allowing other methods to be called at the same time would lead to opening multiple TCP connections at the same time.

Instead, if we create a *WebSocket* API, we only need one TCP connection and, at the same, provide bi-directional communications without additional headers. Moreover, using *WebSocket* allows the application server to send messages asynchronously to the client without the client having to request it, which can be slightly faster than using HTTP which is based on a request followed by a response. Our signaling protocol consists of sending and receiving JSON formated messages, *e.g.* Listing 4.2, over WebSockets by both the application server and the client.

Listing 4.2: General structure of our WebSocket messages

```
1 {
2         "cmd":<cmd>,
3         "data":<data>
4 }
```

When a user enters a group conference, after the page is completely loaded, a WebSocket is created to maintain a connection with our web servers. But before creating the web socket, we must identify the user and check if he has permissions to participate in the conference. The user identification is done by retrieving the session id from the cookie provided by the user-agent (web browser) through the HTTP headers. The web application server retrieves all the information needed from the database in order to check if the user has permissions to join that conference room. It is important to save the user identification before the *WebSocket* connection is created because, after the handshake is performed by the *WebSocket* protocol[7], the HTTP context is lost.

When the connection is established between the application server and the client, a *PeerConnection* is created on the client and immediately after, an WebRTC endpoint is created on the server, specifying a possible set of ICE servers to connect.

At this stage, the web application's user is asked if he wants to share its camera and microphone, share screen or just receive streams from the server. If the user decides to perform a screen share, *adapter.js*[3] may ask to install a plug-in if the browser does not support screen sharing [4]. If the user decides to share either from camera or screen, *getUserMedia* is called with the correspondent constraints in order to obtain a local stream. We use the constraints presented on Listing 4.3.

[2]Although two ICE servers are shown they are, in fact, the same. Showing just one ICE server would be difficult to draw.
[3]`https://github.com/Temasys/AdapterJS`(accessed March 15, 2016).
[4]`http://iswebrtcreadyyet.com/` (Accessed May 11, 2016)

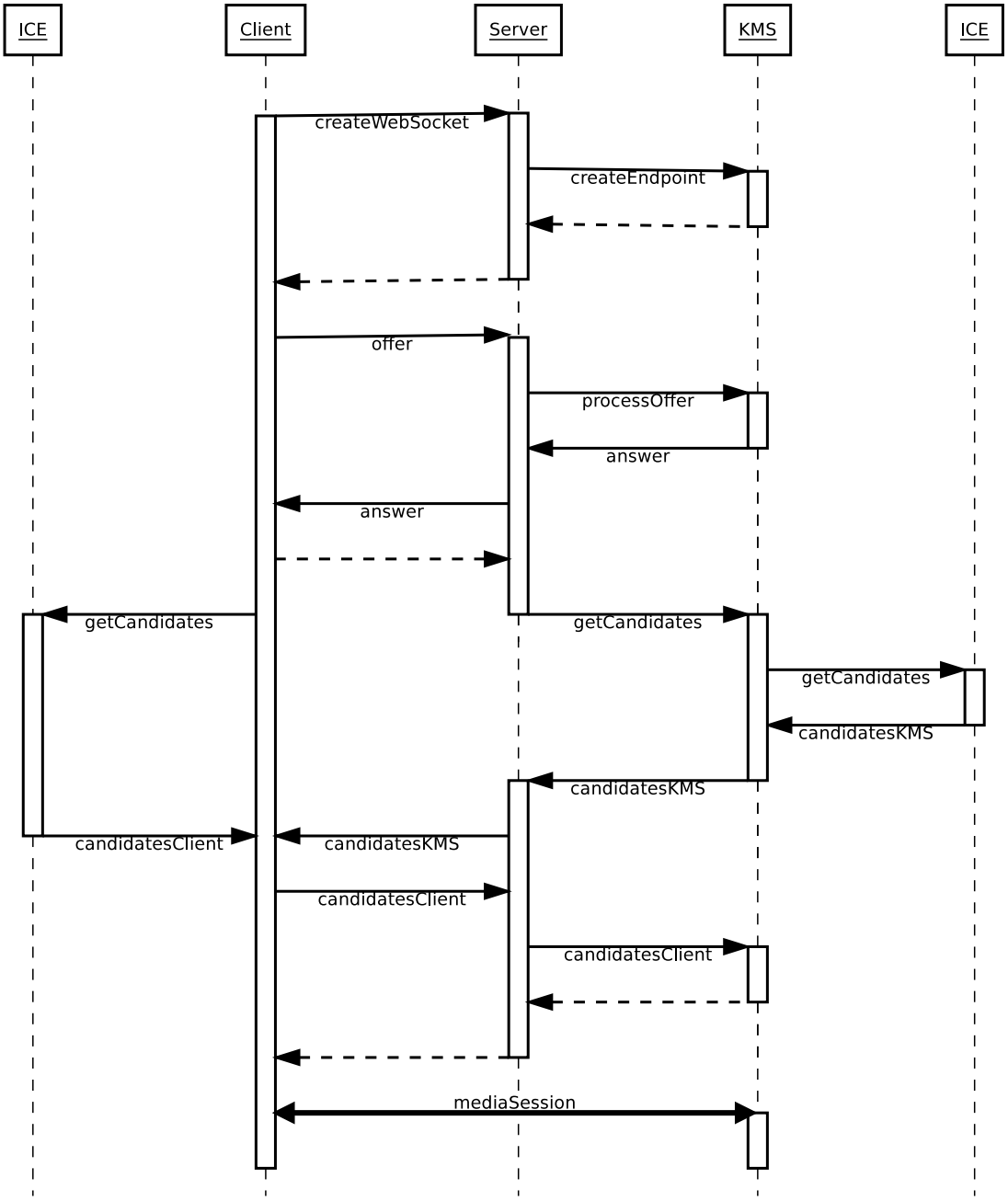


Figure 4.1: Signaling sequence diagram

Listing 4.3: Media constraints

```
var screenShareConstraints = {
        "video": {
                "mediaSource": "window" || "screen"
        },
        "audio": false
};
var cameraMicrophoneConstraints = {
        "audio":true,
        "video":true
};
var receiveOnlyConstraints = {
        "offerToReceiveAudio":true,
        "offerToReceiveVideo":true
};
```

If the user decides to share his camera or screen, a pop-up is raised in order to ask the user to give permission to share those resources. If the resources are shared successfully, the user agent creates an offer like Listing 4.4, sets a *local session description* to its *PeerConnection* and sends it through the WebSocket to the Application Server. If the resources are not shared or the user specified to receive stream only, an SDP offer is created specifying the constraints for receiving only video and audio.

Listing 4.4: Offer created by client

```
{
        "cmd":"offer",
        "data":{
                "type":"offer",
                "sdp":<sdp>     // omitted for brevity
        }
}
```

The *local session description* contains the session identifier, codecs, containers, transport protocols and ports used per media type. The *local session description* is useful to conclude if the client is receiving only, which means that KMS does not need to mix, record nor analyze streams coming from the user's WebRTC endpoint.

The server receives and processes the offer and sets the *remote session description* to its client associated WebRTC endpoint. Then a *local session description*, like the one presented in Listing 4.5, is created on the server and sent back to the client. After that, the server tries to gather ICE candidates.

Listing 4.5: Answer created by KMS

```
{
        "cmd":"answer",
        "data":{
                "type":"answer",
                "sdp":<sdp>     // omitted for brevity
        }
}
```

The client receives the server answer, sets the *remote session description* and gets the ice candidates from the ICE server.

Subsequently, after a while both the server and client receive the ICE candidates that allow the client to connect directly to KMS and vice-versa. The candidates are received at the client which sets them to its *PeerConnection*. The same is done on the server which receives the ICE candidates like on Listing 4.6 from the client and propagates them to KMS.

Listing 4.6: ICE candidates sent by KMS and client

```
{
        "cmd":"iceCandidate",
        "data":{
                "sdpMLineIndex":0,
                "candidate":"candidate:15 1 TCP 843056127 146.193.224.20
                    48828 typ srflx raddr 192.168.1.105 rport 48828 tcptype
                    passive",
                "sdpMid":"audio"
        },
}
```

An ICE candidate contains an IP, port, used transport protocol and an attribute named *sdpMLineIndex* that is used for mapping to the *remote session description* media type. In addition, both intervenients test the connectivity of each ICE candidate. When a connection is established, the user and server start to interchange stream data but other ICE candidates may arrive with better connections. When that happens, the connection changes seamlessly.

Having the media session established, the server starts to record any received stream and the client creates an URL correspondent to the stream location.

4.4 Stream Recording

Initially we experimented with local recording and synchronizing with our servers. Even thought it worked, it consumed too much bandwidth. We then tried recording on server side. In this section, we describe the several techniques evaluated.

4.4.1 Client side recording

In order to record user shared streams, our first approach consisted in locally recording video and audio streams into blobs of limited duration. Because each user was recording directly from their shared local resources, we achieved the best stream quality.

After recording, each block was uploaded to our server through HTTP, saved into the server's file system, the block meta-data was created and inserted in our database. Each block meta-data contained the file's location for the recorded block, the starting date, duration, user identification and group identification.

When the file was completely saved, the meta-data was advertised to the remaining users. This meta-data was simply used to refresh the user interface and was completely discarded after that, because a huge amount of small blocks' meta-data would use more and more memory as time went by.

The process to play a video was quite simple. The user specified which user and date he intended to play, the server calculated the intersection between the requested date and the block bounds and returned the file to the client. Although this idea was fairly simple, we could not achieve seamless sequential block switching. Downloading the video file always took a noticeable amount of time. To solve this problem, while we were playing a block, the next one was downloading in parallel, so when the current block finished playing we would have an available block to play.

With this technique, block switching became more acceptable, but switching the URL always produced a flash. We solved this problem by having two layers and changing the URL on the back layer. When the front layer froze on the last frame, we changed the back layer to the front.

After we implemented this recording solution using this approach, we tested locally (client and server on the same machine) and remotely. For remote connections we observed a fairly high bandwidth usage, mainly because blocks were both sent and received at maximum quality.

We deemed this high bandwidth usage to be a significant bottleneck on our application and decided to explore other solutions.

4.4.2 Server side recording to file system

In addition to perform as an MCU, KMS also allow recording individual incoming streams and the composition of streams which can be reproduced later using the existing *Peer Connection* and as a consequence making our application more efficient and scalable.

Without additional configurations, KMS allows recording video directly to file system by specifying each file path. This approach makes the file management more complex as file paths are relative to one file system within a specific machine. If we use this approach with multiple instances of KMS in order to scale our solution, we have to associate the machine location to each recording chunk and we also have to take into account replication so that we can provide fault tolerance.

Before recording any type of stream, we had to analyze the user media offer in order to check if a video was really being received by KMS, otherwise if we did not verify the user's offer, the recording video would be black.

The streaming content received on KMS was already compressed due to WebRTC's exchanged quality of service metrics data. As a direct consequence, our recorder solution used on most cases less disk space per block,

but would never use more storage than client side recording. KMS allows recording files using the *webm* container by default but *mp4* is also available.

With server side recording, the user would maintain always the same stream URL even if it is playing real time video or reproducing recorded video. It is KMS that sends different content through that stream. When a user desires to play recorded video, a *webSocket* message is sent specifying the time and the intended user id. The group identification is not sent because it is already associated to the *webSocket*. The server performs the same calculations in order to find a block that intersects the requested time, plays it and when finished, the next part is automatically played without the user intervention.

We still observed differences in image quality when switching parts. That was even noticeable if we set a short block duration. We also noticed a small gap on audio when switching blocks but it was acceptable and speech understanding was not very affected.

When we started implementing our solution, KMS had no support for seeking videos, which meant that blocks would would always start playing from their beginning.

If we choose the closest chunk to play this lead to a playing time error that can be at most half the duration of a block, as it can be seen on Figure 4.2 where the black dot represents the requested time to play. In this case, which is the worst, either $block_n$ or $block_{n+1}$ (with duration d) can be played from their beginning with a maximum error (e) correspondent to half their duration ($\frac{d}{2}$) .

In practice, the maximum playing time error coincides with the block duration because we must show always what the user requests, as it can be seen on Figure 4.3. In this case, if a user wishes to play a time that is close to the end of $block_n$ the maximum playing error (e) is at most the chunk's duration (d) because we have to play $block_n$ from its beginning.

KMS does not support playing video at higher speeds, a useful feature that enables users to watch a video in less time than its duration. For playing video with an higher velocity, we used *ffmpeg*[5] to convert the block into a new video with the desired velocity and seek time. Because the media duration is known, when the video started being converted the headers located at the beginning of the file were already written and that made it possible to stream while converting. Although we implemented a solution that worked, we immediately noticed that *ffmpeg* would take some time to initialize and that lead to pauses between switching parts which represents an undesirable behavior.

Later the *Kurento* team released a version with support for seeking videos.

We have not implemented fast forwarding, as currently KMS is not supporting that. We could implement fast forwarding without real time conversion by creating multiple versions of the same video with different playback speeds after the recording of a block. When a user needed to play, he would also need to specify one of the available speed. We did not followed this approach has it would require larger disk space usage.

[5] `https://www.ffmpeg.org/`(accessed: 17 March 2016)

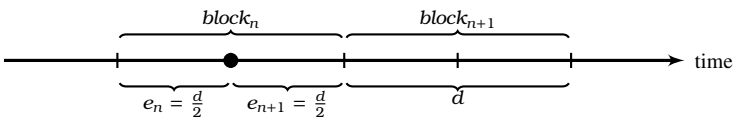


Figure 4.2: Maximum playing error (e) if we play the closest chunk

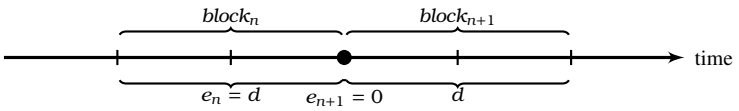


Figure 4.3: Maximum playing error (e) if we play the chunk that intersect the user's time

4.4.3 Server side recording to database

One of our concerns during the development of our solution was the storage scalability. Saving files directly into the file system would require an extra effort to distribute and replicate files among servers. For that purpose, the *Kurento* team developed *Kurento Repository*[6] which is based on *MongoDB* and, as a consequence, improve our solution's scalability by performing replication.

One of the features that *Kurento Repository* provides is the ability to play directly from the database without having to download the entire file to KMS. The same is true for recording, but because the file headers are in the beginning and the file is written until it stops, the headers do not contain the necessary information for seeking the file. Although we gain with scalability with this approach, we lose access over the file for changing it to fit our needs, namely for using *ffmpeg* or other video manipulation tool.

Although we did not implement recorded file seeking, that could be achieved by waiting for full file recording and then proceed to database re-insertion with the correct headers. Another approach would be the specification of the file duration before recording, so the correct file headers could be written a priori. Both approaches were not possible to implement using just the *Kurento* clients. We would need change the source code of *Kurento* in order to add those new features.

4.5 Hyper Content

In this section we describe the algorithm we created to show synchronized interactive content to clients.

4.5.1 Content creation

Our system supports creating superimposed content to video, which is achieved by creating HTML tags on top of the video with the same size. The decision of which content must be displayed to each user is performed by our content scheduler which uses the user's current time in order to synchronize which content is shown or removed from the user interface.

In order to create content, the user has the option to write simple movie captions without writing any code, otherwise, as mentioned before, it can write HTML, CSS and *JavaScript*. The definition of the content's starting and ending time by the user it is not an easy task. Figure 4.4 shows the user interface for creating interactive hyper content manually.

[6] `http://doc-kurento-repository.readthedocs.org` (accessed on 17 March 2016)

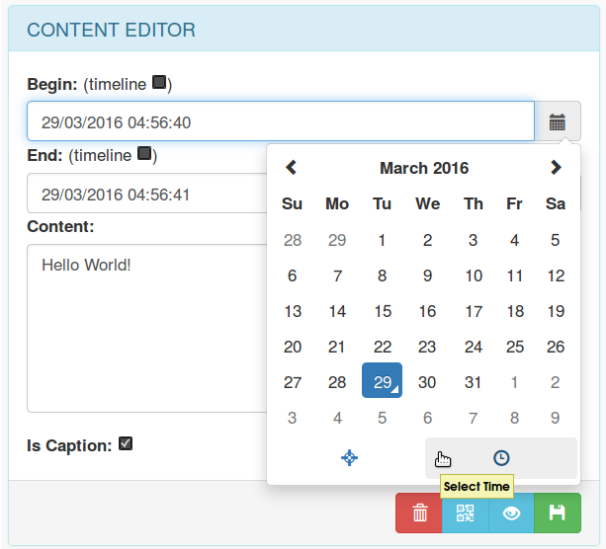


Figure 4.4: Manual content creation

Listing 4.7 shows the structure of the content created by one user

Listing 4.7: Example of content created by one user

```
{
        "cmd":"createContent",
        "start":"2016-03-29T03:56:40.000Z",
        "end":"2016-03-29T03:56:41.000Z",
        "content":"<h1>Hello</h1>"
}
```

Defining the content's time to appear in real time would require a previous user plan. Otherwise the user could make a speech and insert the content later.

As manually content insertion is a laborious task that can be realized after the video is recorder, we provide an alternative mechanism for real time introduction of superimposed content. In order to help the content creator to introduce and synchronize its content in real time, we allow the user to encode its content into QR codes and show it to the camera in real time.

KMS allows registering event handlers for QR code detection. The component that detects QR codes on KMS is called periodically and fires the handlers with the decoded content. Although the QR code detection period is defined by KMS we observed to be one second. This mechanism does not detect if the QR code enters or leaves the screen. We had to implement our own mechanism for detecting those events. Each user session in the server maintains a map with the contents that are present on the screen, which can be represented by multiple QR codes simultaneously. In order to apply our algorithm more efficiently we calculate the content's hash through the *md5* method. If the hash is not present in the map, it means the QR code was entering the screen, we add that hash to the map and associate the current time to it, all the users are notified to watch that content in real time. If the same QR code is not detected after a time bigger then two periods we can conclude that the QR code left the screen and we add the correspondent content into our database. Listing 4.8 shows the pseudo code for QR code leaving

detection.

Listing 4.8: Pseudo code for QR code leaving detection

```
var ongoingCodesMap = {};
var startedCodesMap = {}
function onCodeFound(content) {
        var startingTime = getCurrentTime() - getUserOffset();
        var hash = md5(content);
        if(startedCodesMap[hash] == null) {
                sendQRCodeToEditingUser(hash,content); // insert UI
                startedCodesMap[hash] = currentTime;
        }
        ongoingCodesMap[hash] = currentTime;
        waitTwoSeconds(); // onCodeFound() may be called from other threads
        var newestTime = ongoingCodesMap[hash];
        if(newestTime == currentTime){
                var startingTime = startedCodesMap[hash];
                var endingTime = getCurrentTime() - getUserOffset();
                startedCodesMap[hash] = null;
                ongoingCodesMap[hash] = null;
                sendQRCodeToEditingUser(hash,null); // remove from UI
                insertContentIntoDatabase(startingTime,endingTime,content);
                sendContentToUsers(startingTime,endingTime,content);
        }
}
```

Our main content synchronization mechanism is time based but, with some programming knowledge, it is possible to insert *JavaScript* code that fires events on user interaction. For example, after a teacher's lecture, it is possible to show a quiz to the users in order to understand what they learned and then submit the data to the server for further analysis.

4.5.2 Interactive media synchronization

In order to allow users to visualize past and present communications, we express the user's time position as the time offset between its reproducing time and the current time.

In our solution, content is represented as simple text that can contain HTML, CSS or even *JavaScript*. This content is displayed on defined intervals of time. Multiple contents can be displayed at the same time. In order to achieve that, we define layers above the video with the same size. Each layer is associated to the content. We have taken into account that the amount of content tends to grow with time and the user should only have access to a subset of the content instead of all of it, which would be very inefficient. The content to display for each user depends on the user's position on the timeline. This position is given by an offset between the current time

and the reproducing time. If the user is watching the content in real time this offset is always zero. If the user is reproducing past communications in normal playback rate, this offset is constant and, as a consequence, there is need to re-synchronize this value. If, for instance, we had to implement communication reproduction with a faster or slower playback rate, we would need to take into account the fact of this time offset not being constant.

Each content, that users receive from our application server, is divided into two components, the *start* and the *end*. The *start* component contains a time stamp, the content identification number and the content itself in form of text. The *end* component only contains the time stamp and the content identification number.

The users will receive their contents from the server on five different situations:

- Conference room entrance (advertised by server).
- Set of events empty and server has more content (client requested).
- Content is created (advertised by server).
- Content is removed (advertised by server).
- User navigates to different point in time (client requested).

The content to return is given by the union of the two following content subsets:

- Content that should be currently on display.
- A subset of contents whose starting time is immediately after the user's time.

The second subset is used for predicting which content the user will watch and avoid requests during its events.

The content description messages that are sent to each user contains the constant itself on the form of a list of events and an attribute that specifies if the server has more content to return.

Listing 4.9 shows the structure of the content sent to users.

Listing 4.9: Exampe of content sent to users

```
{
        "cmd":"content",
        "data":[
                {
                        "id":"56f9fd1aa986c615fab43d69",
                        "time":"2016-03-29T03:56:40.000Z",
                        "type":"start",
                        "content":"<h1>Hello</h1>"
                },
                {
                        "id":"56f9fd1aa986c615fab43d69",
                        "time":"2016-03-29T03:56:41.000Z",
                        "type":"end"
                }
        ],
        "more":false
}
```

When a user enters the conference room, immediately after the *WebSocket* creation the server sends him the current content. The user receives the content, sorts all components by time and creates a set of events. All the events before the user's time are displayed and removed from the set of events. A timer is scheduled for the first component on the event set and the process repeats while the set is not empty.

If the event set is empty, there are two options. If the server contained more content, a new request for content is made and the process starts from the beginning, namely the server sends the correspondent content again. If the server has no more content, the process is stopped until it sends more.

Listing 4.10 presents the pseudo code for our content scheduler.

Listing 4.10: Pseudo code for hyper content scheduler

```
function scheduleContent() {
    var navigatedTime = getCurrentTime() - getUserOffset();
    while ( hasLocalEvents() ) {
        if ( firstEventIsOlderThan(navigatedTime)) {
            if ( firstEventIsStart() ) {
                addHtmlLayer(event);
            } else {
                removeHtmlLayer(event);
            }
            removeFirstEvent();
        } else {
            break;
        }
    }
    if ( hasNoLocalEvents() && serverHadNoMoreContent()) {
        return; // nothing to do
    }
    if ( hasNoStartEvents() && serverHadMoreContent() ) {
        requestContentFromServer(navigatedTime);
    } else {
        waitUntilNextEvent();
        scheduleContent();
    }
}
```

4.5.3 Content Removal

Each layer is identified by its content identification number which is used by our content scheduler in order to remove disappearing content.

If a user clicks over one layer's content the layer becomes selected (a frame is drawn around it) and its content identification number is added to a list containing the selected layers. When a user wishes to remove superimposed content, the selected layer ids are sent to the application server, with the same format as shown on Listing 4.11, and subsequently all users will receive a message in order to update their user interface.

Listing 4.11: Example of removing content message sent by one user

```
{
        "cmd":"deleteContent",
        "content":["56f9fd1aa986c615fab43d69","57339588ac687b423e5b3a1e"]
}
```

4.5.4 Security Concerns

Our solution is flexible on what kind of interactions are possible to the users in real time, but allowing users to write *JavaScript* that is executed on the other users' browser would allow attackers to misuse their resources and access critical information. We could solve this problem easily by escaping any *script* tag present on the content, but we would sacrifice the kind of interactions that are possible. By not allowing *JavaScript* we would need to implement a subset of conceivable actions a priori and fire them when a type of message is received. We decided to postpone, to future work, the correction of the security vulnerabilities that are exposed by evaluating *JavaScript* because, in order to offer the same interactions, we would have to do an exhaustive functional requirements gathering and, in our prototype, our goal is to explore users behavior when interacting with our system.

Another way to solve this problem, which is not within the goals of this thesis, is to analyze the *JavaScript* code and detect if it is malicious. Although we have not performed a detailed study on this field, we are aware of its complexity and the existence of solutions to deal with this problem, in particular *EarlyBird*[36] came to our attention.

EarlyBird uses machine learning techniques in order to improve malicious *JavaScript* code detection accuracy.

4.5.5 Time Manipulation

We have used *vis.js* to display out timeline. This library was created for content navigation through time, but it was not designed to be always moving automatically. We have created a background timer that performs the animation of moving the window of time bounds and user navigated time marker.

Figure 4.5 shows the graphical appearance of our interactive timeline. In order to navigate through time, the user must drag and drop the timeline horizontally. When the user drops the timeline an event handler is called with the new user's time offset, which will be used to send a message to the server in order to choose the correspondent content to display.

We have noticed the server time being different from client time and this created timing inconsistencies, such as showing existing recorded blocks of movie in the future. To solve this problem, the server sends its time to the

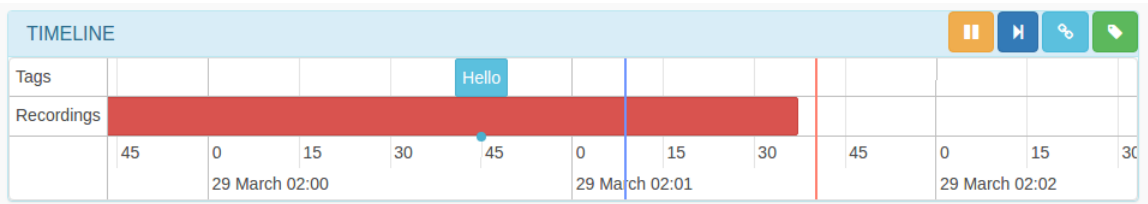


Figure 4.5: Interactive timeline

client immediately after the *webSocket* creation. We synchronize the timeline with the server and although it may exist a small error due to network transmission time, the graphical error is negligible.

4.5.6 Time annotations creation

Time annotations are a simpler way to mark points in time and share them with other users. When a user enters a conference room, the server sends all annotations so they can be displayed directly into the user timeline. When each time annotation is created, all users are notified so they can update their interfaces.

Figure 4.6 shows the creation and placement of annotations on the timeline. To save the annotation the user must click on the floppy icon in order to save it and notify the conference participants.

Besides the ability to create tags, it is also possible to create time hyper-links, see Figure 4.7, that can be sent to other users externally so they can navigate directly to the content when entering the conference room.

4.5.7 Content Search

Users can search for annotations and contents and travel to their correspondent times. In the case of hyper content, after handling the result from database, we extract the text by discarding HTML tags with *Jsoup*[7] and apply the query again. We extract the text from HTML because accidentally searching for text contained in HTML tags would lead to incorrect results.

Figure 4.8 shows how the search results are displayed to the user. Each result entry contains an icon specifying the type of result (hyper content, time annotation, ...).

4.5.8 View composition and switching view

In order to show all streams to all users, each conference participant could receive individual streams for each user that share video. If there are n active participants in the conference room, the number of streams per user is n (1 for sending video and $n - 1$ for receiving) and, consequently, the number of streams in that conference room, as observed in Figure 4.9, is n^2.

By mixing all streams into a single one, users can receive a single stream with the image (in a grid) and audio of all active users. With this approach, the total number of streams within a conference room can be, at most, $2n$ (each user has 2 streams, one to send and another to receive) as observed in Figure 4.10.

The numbers we have presented above were calculated taking into account that all users would receive and send video at the same time, which is the worst case scenario, although users may choose just receiving streams.

[7] `http://jsoup.org/` (Accessed 21 March 2016)

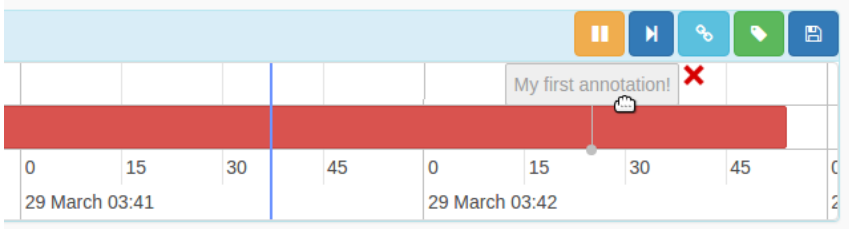


Figure 4.6: Creating a time annotation

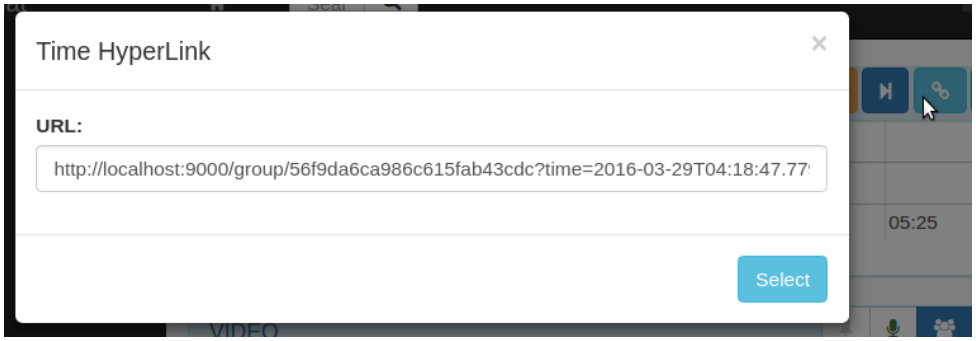
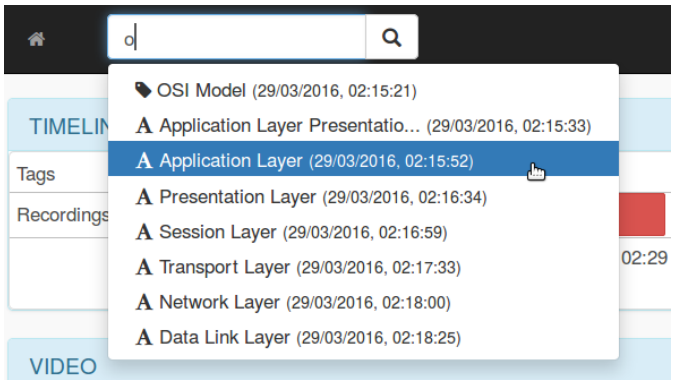
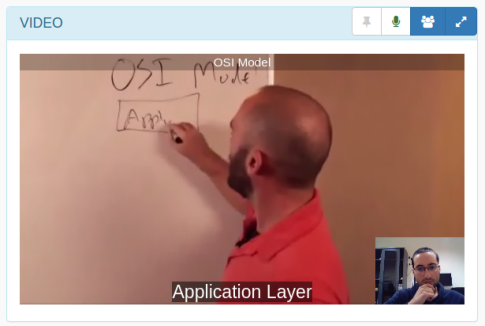


Figure 4.7: Time link creation



a) Result list



b) Selected result

Figure 4.8: Example of search results

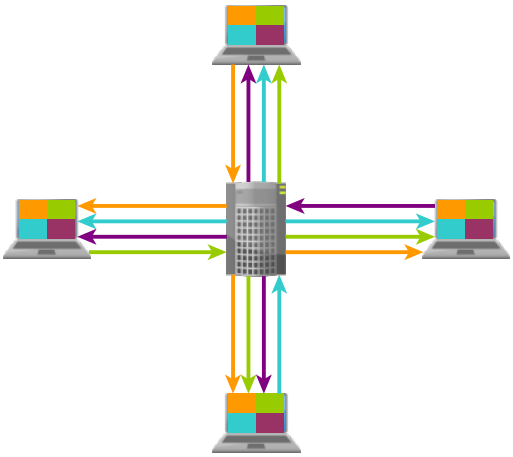


Figure 4.9: Client-side stream composition

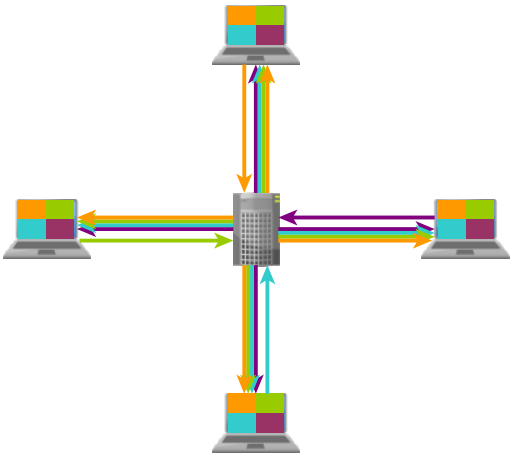


Figure 4.10: Server-side stream composition

When the amount of participants is just two, both solutions require the same amount of streams but as the number of participants increase, the composition operation tends to be a more efficient approach.

The composite operation, as shown on Figure 4.11, is performed at KMS and works by decoding the incoming client streams, mixing audio and video and encoding again to send back to users.

We provide a way for users to select the composite view of the conference room or the video shared by a particular user device as it can be seen on Figure 4.12.

When a user plays or ends playing a block of recorded video, it requests a list of available devices from each user that can be played at the navigated time by querying the database.

On the other hand, if a user changes to real time, the list of devices is extracted from the set of *webSockets* that are associated to the respective conference room. This list of devices is also sent to the users that are also in real time mode whenever a new user enters or leaves the conference room.

In addition to select a stream to view, which remains the same if not interrupted, we also provide an automatic mechanism for switching the view to the current speaker. With respect to sound analysis, the sound samples are analyzed on the client side through the web audio API.

Our speech detector is straightforward, we could perform a spectral analysis in order to understand if the analyzed sound contains frequencies in the range of the human voice but instead we just capture sound samples in real time and calculate the maximum sound amplitude. If a sound sample has an amplitude bigger than a factor of the maximum amplitude (we have used an empirical value of 10%) we say that the user is speaking and therefore we send a message to the server if the speaking state has changed. Subsequently, the server receives the user speaking state and sends it to the other users so they can request a different view.

If multiple participants are talking at the same time, the selected view will change indeterminately between them. Although such behavior is not the most desired, we could have done a dedicated composite view in order to show the talking participants.

4.5.9 Non interactive media synchronization

By default, each user receives the real time mixed video and audio of every users. As described on the previous section, users can switch the current view even for recorded media.

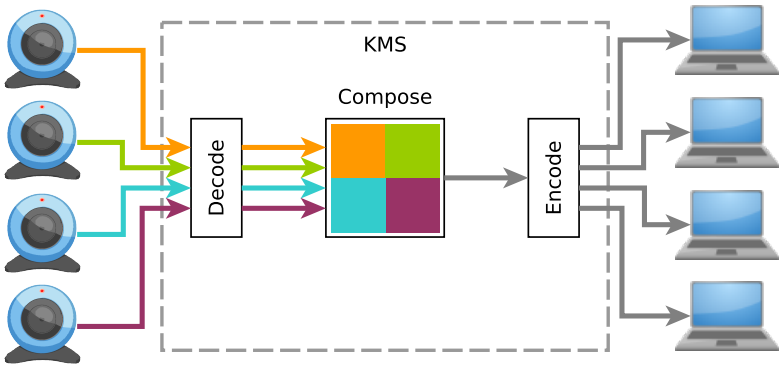


Figure 4.11: Stream decoding, composition and encoding

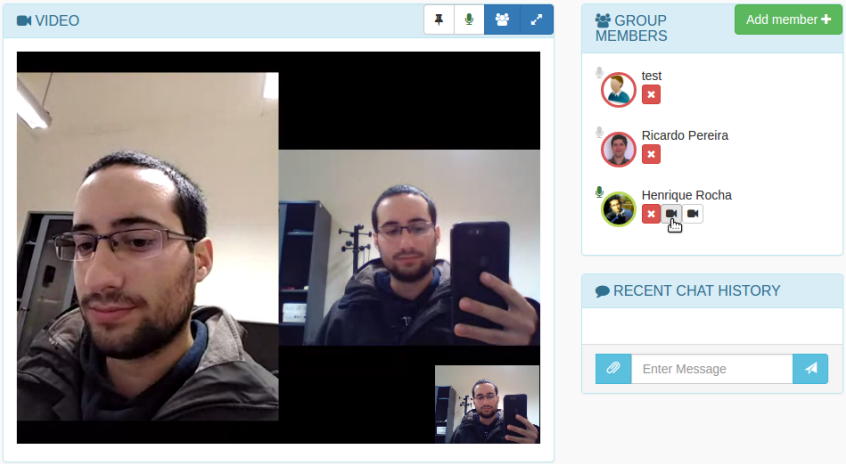


Figure 4.12: Multiple devices per user

Whenever a user changes the current viewed time, our server calculates which chunks to reproduce.In order to determine which chunks are going to be played we could find any chunk that matches the session id and intersects the given user time. Although this seems logical, we limit the size of the results to one and ignore the starting component of the recording chunk. As a consequence, we will obtain the recording chunk that ends right after the playing time.

The obtained chunk may or not intersect the given time. If it intersects then we immediately start playing the chunk, otherwise we know that the chunks starting time is after the playing time and we schedule the player to reproduce after a time duration given by the difference of the playing time and chunk starting time.

In order to reduce the delay between switching parts we perform a query on the database for finding the next chunk to play while the selected media is still playing. In order to find the next chunk, we use the chunk that has a bigger id immediately after the playing chunk.

In order to allow the users to watch an individual image and ear the audio of everyone, we are using two players, one for audio and another for video.

If a user leaves a conference room, it is expected that the sequence of played blocks ends. When a block is not found for a given *webSocket* session, we try to find a block for the same user and if not possible, we try to play a composite recorded block, lastly, if every attempt fails we wait for the next interval that contains media.

4.5.10 Chat

Not less important the chat functionality was implemented using *webSockets*. When a message is sent by a user, it is received by the server, which records it and sends to all the users.

In order to support receiving a huge amount of messages and not having a bottleneck when loading the conference room, we just load an amount of messages that fill the chat panel. When a user scrolls the chat to read old messages, we detect when the chat scroll bar reaches the limit and request older messages from the server automatically.

Moreover, when the clients receive messages, whether older or newer ones, their ids are used to place them in the correct position. In other words, older messages are placed in the top of the chat panel and newer ones in the bottom.

Besides that, we also support sending files to other users, which are stored on the database and accessed through an URL.

4.5.11 Collaborative editor

Our collaborative editor is a simple text editor that is synchronized with all participants within a conference room implemented using *ot.js*.

The state of our collaborative editor is not saved on the database every time it changes. Instead, the users just synchronize the editor content among themselves using the application server to relay editor changes and save on demand.

When a user joins a conference room and the content differs from the version stored in the database, our server does not have the right content to deliver the user. As a consequence, our server names a random participant to yield its collaborative content and delivers it to the joining user.

All operations that we perform over the conference room's collaborative editor are performed in mutual exclusion so all updates are delivered in the same order to all participants. For example consider two write events W_1 and W_2 if they occur at the same time, our server will receive one of the events first, say W_1, and delivers is to every users including its writer. Only then the server delivers the W_2 content. With this serialization mechanism we can ensure the content is equal on every user editor.

According to *ot.js* there are three types of operations:

- retain → advance the cursor by a given number of positions.
- insert → write the given text in the current position.
- delete → delete a given number of characters forward.

In summary, our server just ensures the ordering of operations and delivers the messages among users without even parsing the message's content.

4.6 Chapter Summary

In this chapter we have described how we implemented the various components present on our architecture, the challenges that we have faced and the solutions we have found.

We started by defining the database model, which influenced directly our system's behavior and the extensibility to new functionalities.

Then, we underlined the requirements of our signaling protocol and, as a consequence, we have described, in detail, the protocol itself.

With the signaling protocol implemented, we observed the first outcomes of using WebRTC, the basic functionality we have implemented was an echo of the video and audio sent by users. However we have implemented other features such as switching to other user streams, recording and mixing multiple streams into a single one.

In respect to hyper-content, we have described how to create and search for content either being superimposed content to video and time annotations. In relation to displaying content to users we have also defined how we have synchronized the content to show and the security concerns of our choices.

Then, we have described how we have implemented our chat and collaborative editor.

Chapter 5

Evaluation

In the previous chapter we described how we implemented our solution namely the data model that we used, the description of the signaling protocol, how we performed stream recording, how we overlayed interactive content to video.

In this chapter we describe how we deployed our solution, tested our prototype, show and analyze the results in order to validate the contributions of this thesis.

5.1 Solution deployment

In this section we describe the hardware and software we have used to deploy our solution.

5.1.1 Hardware

The hardware was gently provided by INESC-ID[1]. The specifications of the server that we used are described in Table 5.1.

5.1.2 Operating System

On Table 5.2 we present an overview of the operating system configurations of the machine we used for deployment.

[1] http://www.inesc-id.pt/ (Accessed March 26, 2016)
[2] http://www.supermicro.com/products/system/2U/6027/SYS-6027R-72RF.cfm (Acessed March 26, 2016)
[3] https://www.debian.org/ (Accessed March 26, 2016)

Table 5.1: Hardware specifications

Server	Supermicro SuperServer 6027R-72RF[2]
CPU	2 x Intel Xeon E5-2640V2, LGA 2011, 2.0GHz, 8C/16T
RAM	8 x DDR3 REG16G-1600DDR3, 16GB, DDR3-1600, Registered ECC, memory
Network cards	2 x Intel Corporation I350 Gigabit Network Connection (rev 01)
Disks	2x SAMSUNG SSD 840 PRO 256GB SATA III (Drive 0 - RAID 1 - 237.486 GB) 4x WESTERN DIGITAL 3TB SATA III 64MB RED (Drive 1 - RAID 5 - 8.185TB)

Table 5.2: Operating system specifications

Kernel	Linux version 3.16.0-4-amd64
Distribution	Debian GNU/Linux 8 (jessie)[3]
Swap	16GB (Drive 0)
Root	40GB (Drive0)
EFI	200MB (Drive 0)
Bcache	Remaining space (Drive 0 & Drive 1)

5.1.3 Software

INESC-ID provided us a restricted *Linux* account without administration permissions, which prevented us from installing our solution directly on the machine because we would need administrative privileges to install all the software that our solution depends on. Despite that limitation, they provided access to Docker[4] which run on the host operating system as an isolated process in user space.

Inside a Docker container we have the administrative privileges to install all the software dependencies.

We could use one docker image for each component but, in order to reduce the network usage, we preferred to install all the components within the same image. At the same we provide an all-in-one easy to deploy solution as all the IPs are local.

We present, on Table 5.3, the software we have installed inside our docker container including the versions that are in use.

As we provide an application compatible with most modern web browsers, the client application runs on each user's machine.

5.2 Tests Objectives

We have tested our solution with real users for a better understanding of their difficulties and what can be done in order to improve our solution's user interface usability.

We have also tested the performance of our solution by measuring the used resources. Those performance tests are crucial to ensure that our solution is in fact stable and users can use it endlessly without decreasing the quality of their experience.

[4] `https://www.docker.com/` (Accessed March 27, 2016)
[5] `http://www.ubuntu.com/` (Accessed March 27, 2016)

Table 5.3: Installed Software

Name	Version
Ubuntu Server[5]	14.04 LTS
Oracle Java	1.8.0_77
Play Framework	2.5.0
MongoDB	3.0.10
Kurento Media Server	6.4.0
Kurento Repository	6.3.1
Python	2.7.6

5.3 Performance Tests

In this section we describe performance tests that we have applied and their respective results.

5.3.1 Tests Scenarios

In order to benchmark our system, we have implemented a small *Python* script using *psutil*[6] that collects to a text file with a periodicity of one second: CPU, RAM usage relative to each running process and network usage relative to each interface.

We would like to collect network information relative to each process, which *nethogs*[7] provide but it could not capture the network usage of some processes. Although this would be useful for a deep analysis, we know which network interfaces are used to establish connections between processes as we show on Table 5.4.

The performance test scenario that we have defined consists of two phases, the first phase consists only on having users, with similar computer and network specifications, entering sequentially on the conference room. The second phase consists on the users leaving the conference room. Each event, joining and leaving, occurs with intervals of one minute in total of 13 minutes (780 seconds).

5.3.2 Test Results

In this section we discuss the results of our performance tests performed after implementing our solution.

Network usage

Every time a user shares its camera or its screen in the context of a conference room, its offered video is sent to the server and, independently from watching an individual stream or the mixed version, the server just sends a unique stream back to the user.

From the media server perspective if there are n clients connected, each of them sending and receiving one stream, it is expected that server sends and receives also n streams. As such, we expect that the amount of network traffic increases linearly as users join a conference room. Figure 5.1 confirms our expectations. Each vertical yellow line represents one event: the first seven events are users entering the conference room, the next ones represent users leaving the conversation.

The blue peaks are caused by the signaling phase and web page downloads, including resources such as images, stylesheets and javascript files. The green peaks are caused by video and audio being transfered between KMS and

[6] `https://github.com/giampaolo/psutil` (Accessed March 27, 2016)
[7] `https://raboof.github.io/nethogs/` (Accessed March 27, 2016)

Table 5.4: Mapping of connections by network interface

Interface	Connection
Loopback	Web Server ↔ MongoDB
Loopback	Web Server ↔ Kurento Media Server
Loopback	Kurento Media Server ↔ Kurento Repository ↔ MongoDB
Ethernet	Web Server ↔ Client
Ethernet	Kurento Media Server ↔ Client

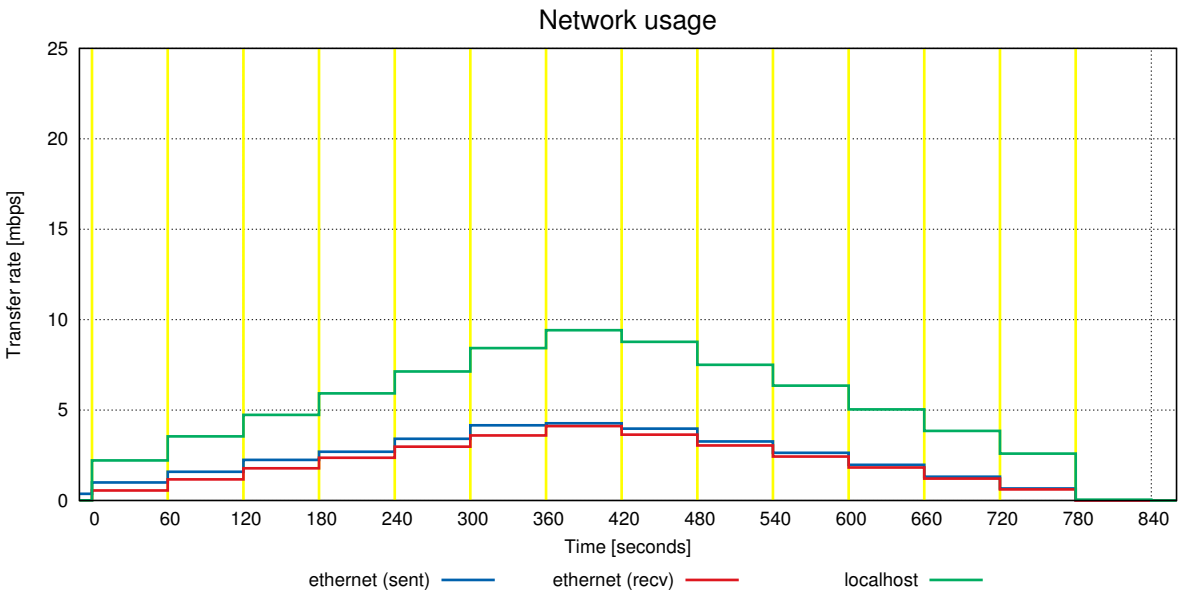


Figure 5.1: Network usage after implementing all features

MongoDB through *Kurento Repository*. Each peak occurs every time a block of video is recorded, which in this case is every ten seconds. The recordings are synchronized so all user and mixed blocks start and end at the same time. That is why the amount of work done every ten seconds accumulates, and because this is performed locally, the maximum transfer rate is limited by the performance of the memory as buffers are written to buffers then to disks. Sent data transfer rate has no significant peaks as HTTP requests and signaling information contains little information.

Figure 5.2 shows the average transmission rate per interval of consecutive events.

If we consider all streams equal and an incoming stream uses $x$kbps, with n incoming streams the rate of data received at KMS is nx and the expected rate of data transfered to *Kurento Repo* using localhost is $(n + 1)x$. Due to the data being transfered also from *Kurento Repo* to *MongoDB* via localhost, the expected total rate of data transfered through localhost is $2((n + 1)x) = (2n + 2)x$, which explains the reason for the average amount of data transfered through localhost being more than twice the average amount of data received from users. With this results we conclude that if we want to scale our storage solution using the *MongoDB*'s cluster configuration, both *Kurento Repository* and KMS should be installed on the same machine, as depicted on Figure 5.3, because the loopback interface can handle bigger transfer rates than the remaining network interfaces. Installing the repository on the same machine as a database node does not ensure that recorded videos are stored in the same machine, for this reason we would prefer installing KMS and *Kurento Repository* on the same machine.

On the other hand, Figure 5.4 shows the respective network usage on the client side during our test case. We observe that in the first seconds the client adjusts the video quality it sends to KMS. Whenever a new client enters the conference, we observe that KMS decreases the video quality in order to instantaneously integrate a new user into the conference room. After a while, KMS realizes that the network can handle the increase of clients and sends the video with a better quality to every participant. When a user leaves the conference room KMS has no need to decrease the participant's video quality as less network bandwidth will be used.

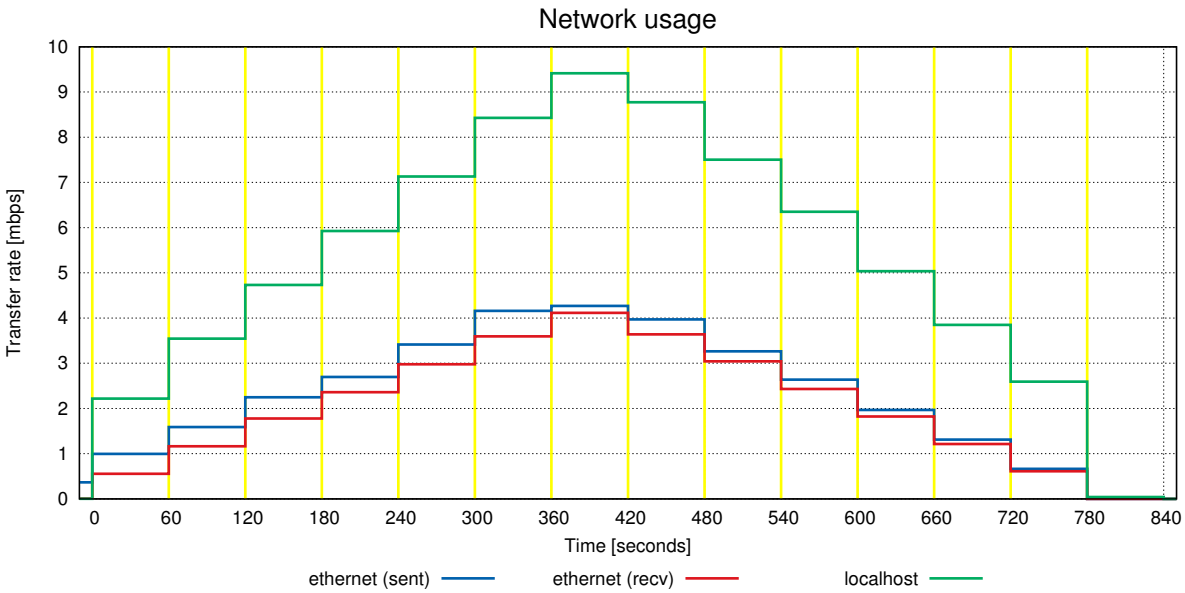


Figure 5.2: Average network usage per interval of events after implementing all features

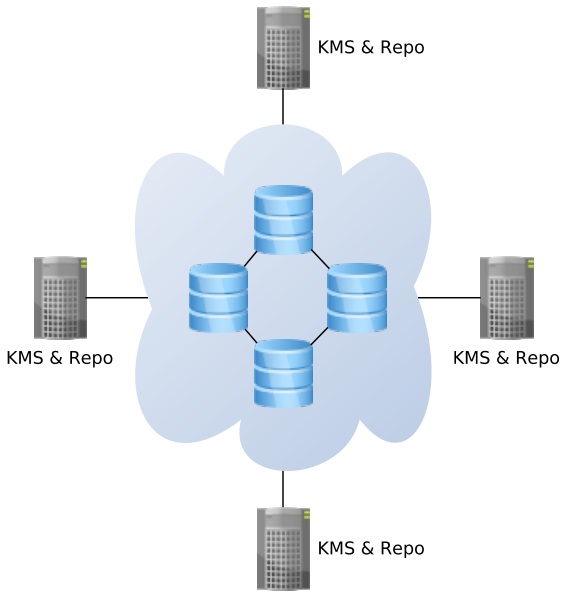


Figure 5.3: Repository and KMS in the same machine

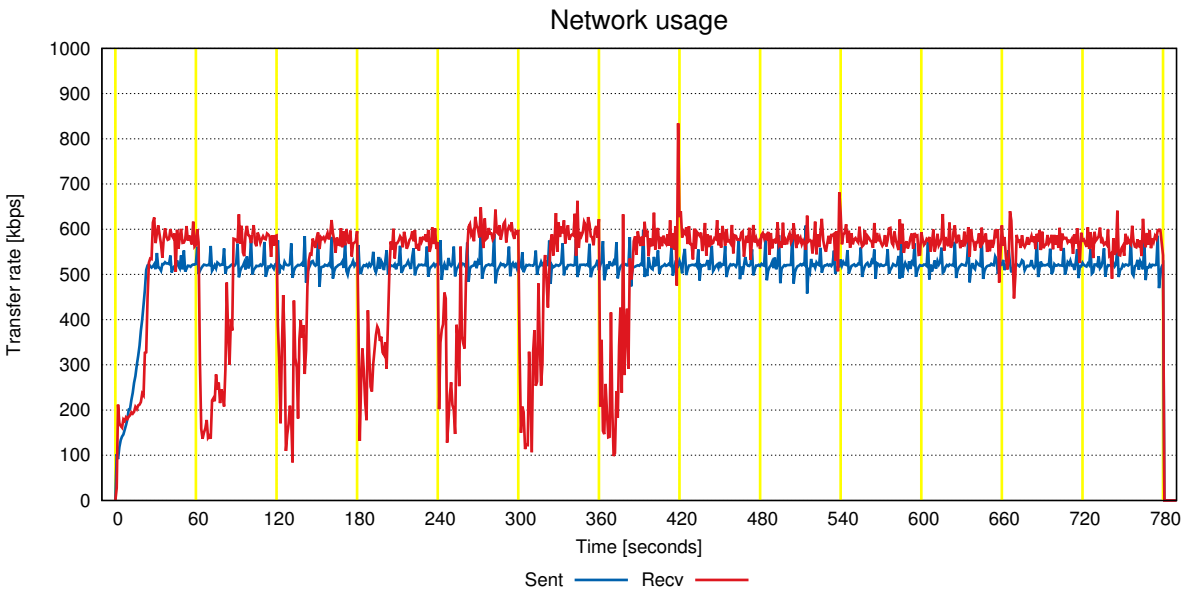


Figure 5.4: Client Network usage during our test case

Memory usage

Figure 5.5 shows the memory usage during our performance test. Both Java virtual machine (JVM), *MongoDB* and KMS performs their own memory management by holding and recycling objects when needed. The expected and observed behavior of the memory usage is growth of memory usage while the users are entering the conference room and a memory usage stabilization afterwards.

MongoDB memory usage keeps increasing because it tries to fit part of the database on Random-access memory (RAM) for fast read access. *MongoDB* checkpoints data to disk every 60 seconds or when journal data exceeds 2GB[8], that explains the small memory usage peaks during our test case. When the conference room is empty there are no video recordings, which explains the memory stabilization at the end.

KMS memory usage was increasing and in fact we did not expected that. We have done more tests and we have observed that by disabling the recorder feature, memory usage kept linearly related to the amount of clients present on the room. Therefore, we have verified our implementation and we have concluded that nothing was wrong with our code: we have confirmed that every allocated resource was indeed released after the end of recording. As a result of our intensive tests and consecutive failures we suspected that the problem was not ours.

Hence, in order to confirm our suspicions, we have decided to read the implementation of KMS and we have found that KMS was not releasing the memory if the recorder was stopped by us. Subsequently, we have changed the source code and submitted to *Kurento* team through a *GitHub*'s *"pull request"*, which was accepted.

Figure 5.6 shows the memory usage during our performance test case using the patch that we applied to KMS. As we can observe, there were KMS and also *Kurento Repository* memory usage improvements.

CPU usage

Figure 5.7 shows the percentage of CPU usage during our performance test case. Each 100% represents one CPU core, although that does not mean one CPU is fully used, for example two cores at 60% represent 120% CPU

[8] `https://docs.mongodb.org/manual/faq/storage/`(Accessed March 28, 2016)

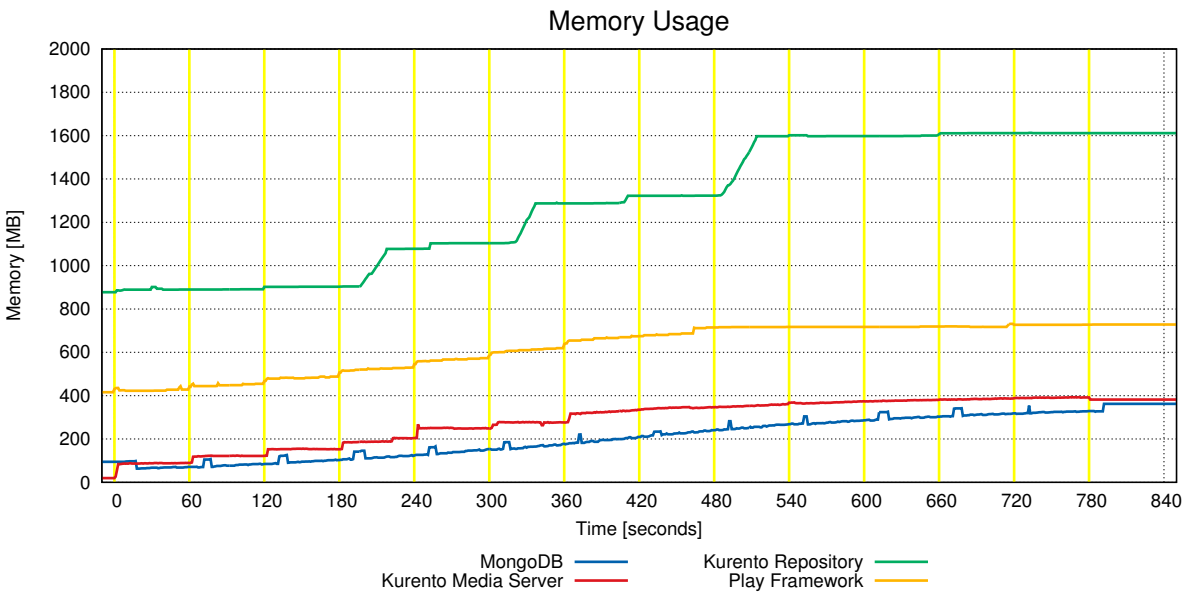


Figure 5.5: Memory usage after implementing all features

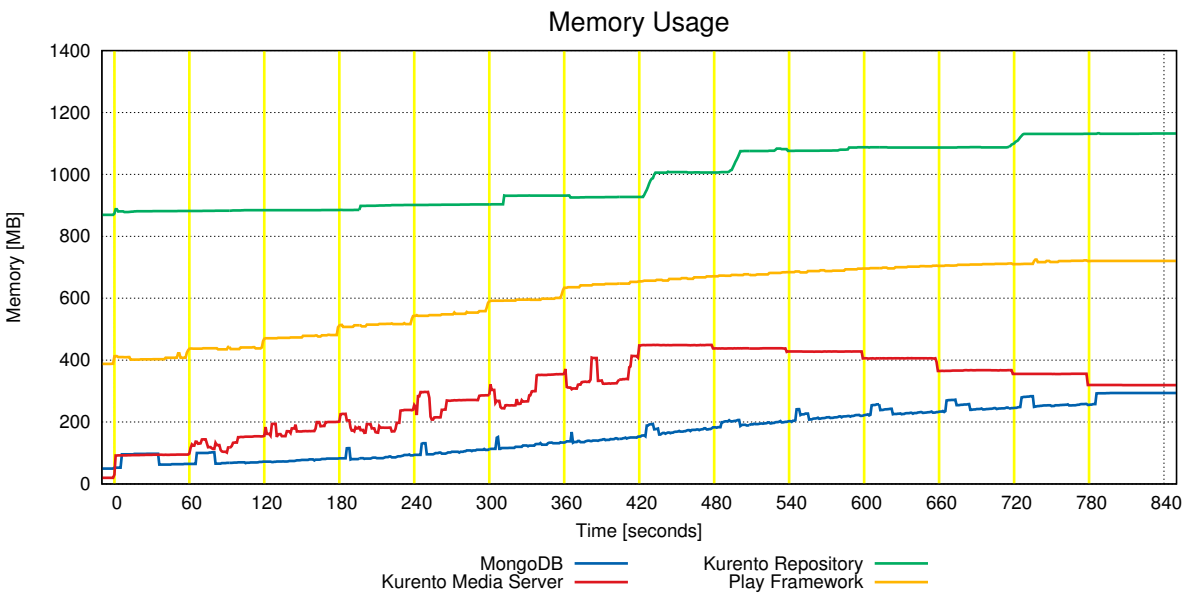


Figure 5.6: Memory usage after fixing recorder memory leak

usage. As we can see, the percentage of CPU used increases and decreases linearly in function of the amount of conference participants.

Figure 5.8 is the same as Figure 5.7 but zoomed over *MongoDB*, *Java* and *Kurento Repository*. As we can see there are periodic processing peaks every ten seconds that are due to block recording. KMS is responsible for most CPU usage.

Figure 5.9 shows the average used CPU per interval of consecutive events.

Just for testing purposes, we performed the same performance tests disabling QR codes detection in order to understand how CPU intensive this task is. Figure 5.10 shows the results for the same experience but with QR code detection disabled.

We conclude that QR code detection is a very intensive task, approximately doubling the amount of work performed by the CPUs. Without this feature, the network and memory usage had insignificant changes compared to CPU usage.

Even though, with this test results, we conclude that our solution's bottleneck is the CPU usage at KMS.

Two consecutive test cases

We have also performed two consecutive test cases in order to understand if we the influence of the first test case over the second, as observed in Figure 5.11, there is no CPU influence between two test cases.

On the other hand due to memory recycling techniques we can observe in Figure 5.12 that some of the memory that was allocated in the first test was reused in the second case.

One hour test case

Figure 5.13 shows the memory used by our solution with seven clients simultaneously on the same conference room during one hour. As we can observe, KMS memory usage stabilizes.

5.4 Usability Tests

In this section we describe usability test scenarios that we have applied and their respective results.

5.4.1 Tests Scenarios

In order to evaluate the usability of our solution, we have performed usability tests with the help of real users with different backgrounds and ages.

We handed a guide to the users with five tasks to perform. The metrics we used for each task were: number of clicks, number of errors (including a description) and time spent.

The tasks we asked the user to perform were:

1. Login into the system with the provided credentials and accept the received friendship request. Find a given user by its name and add him as a friend.

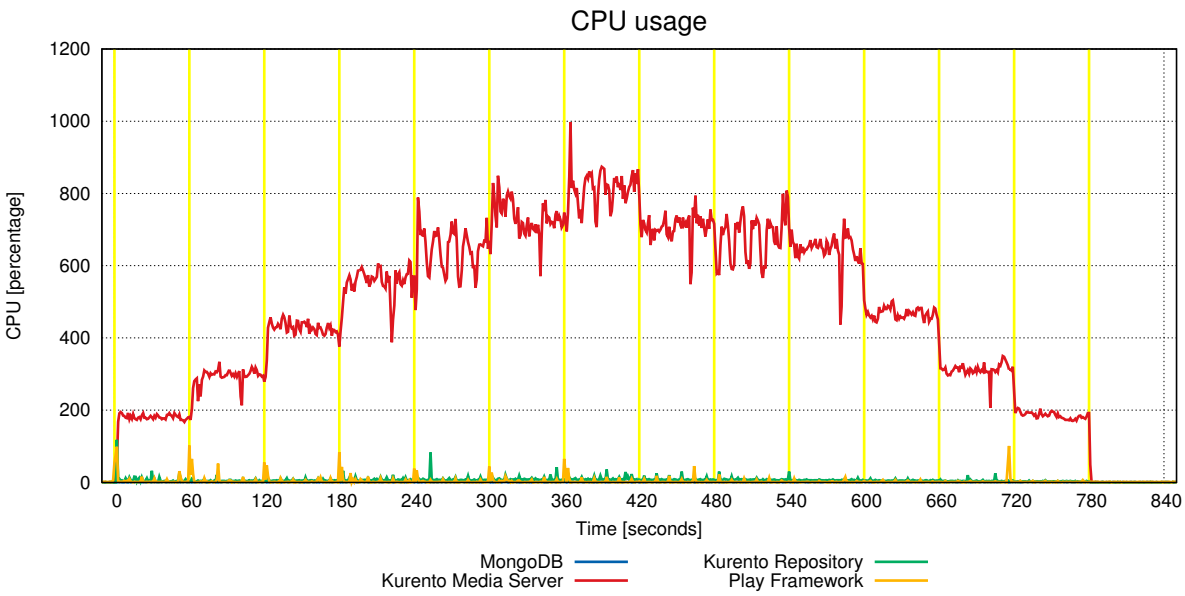


Figure 5.7: Percentage of CPU used during the performance tests

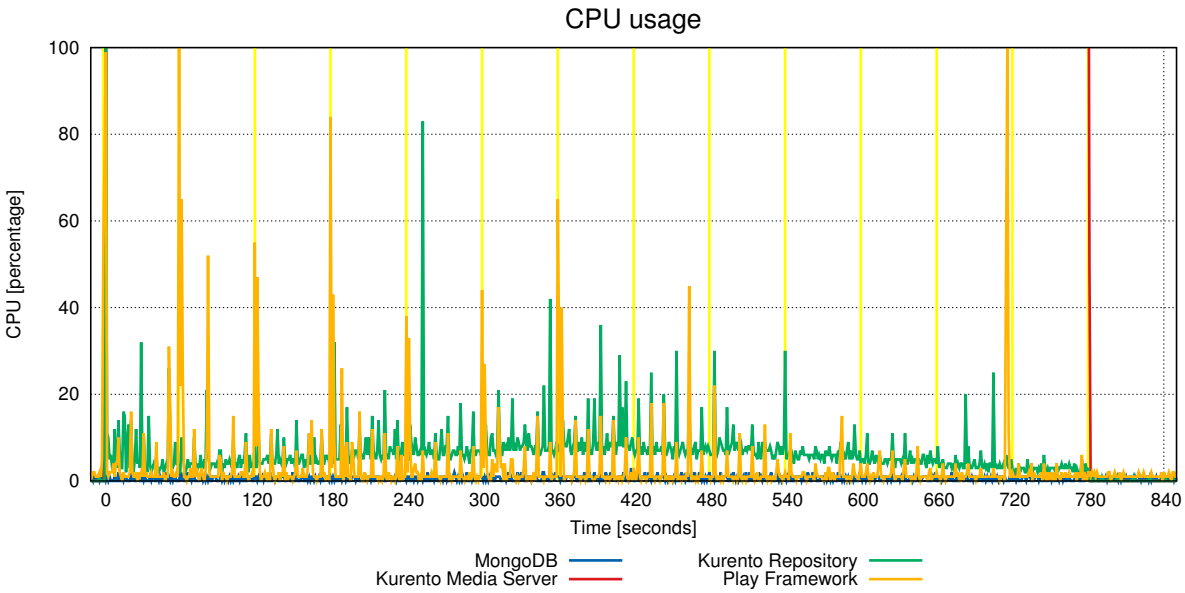


Figure 5.8: Percentage of CPU used during the performance tests (zoomed)

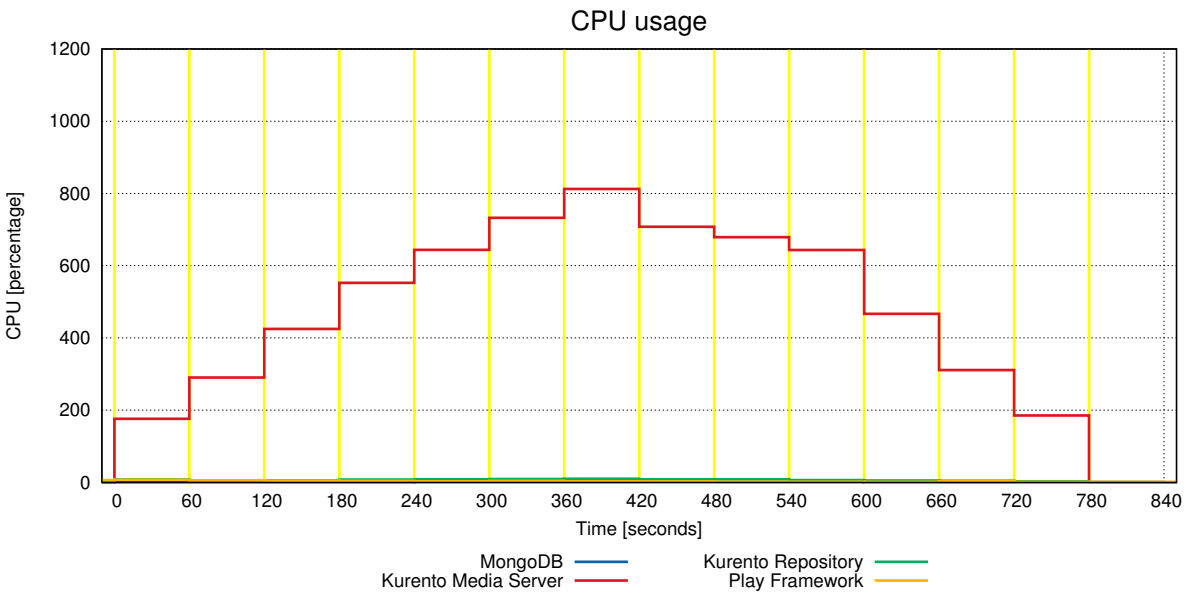


Figure 5.9: Average percentage of CPU used per interval of events after implementing all features

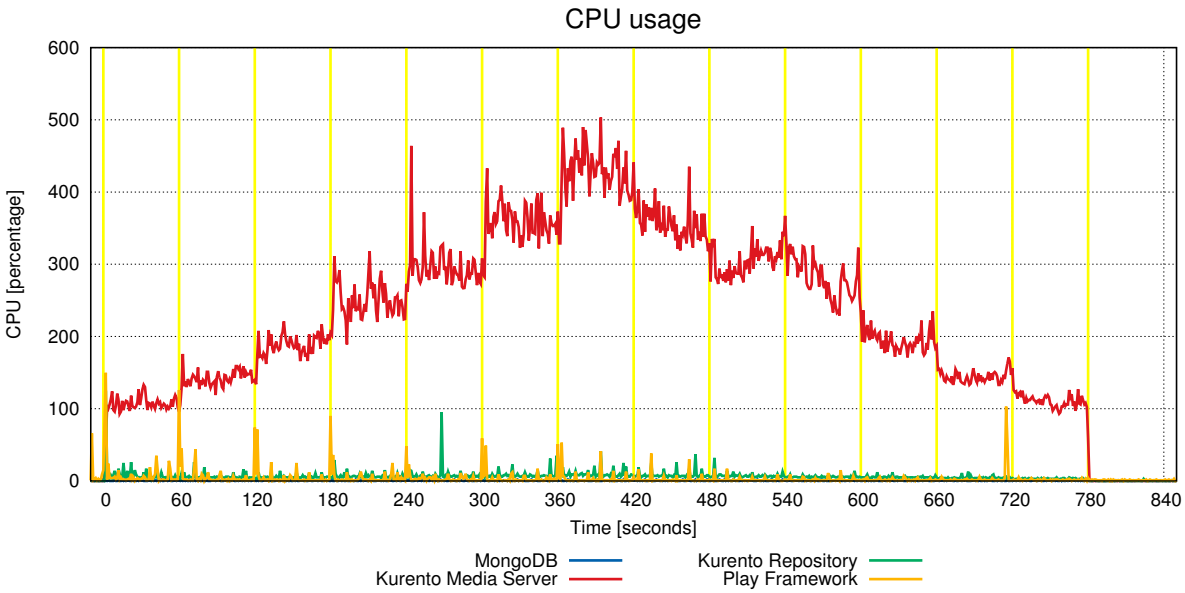


Figure 5.10: Percentage of CPU used during the performance tests without QR code detection

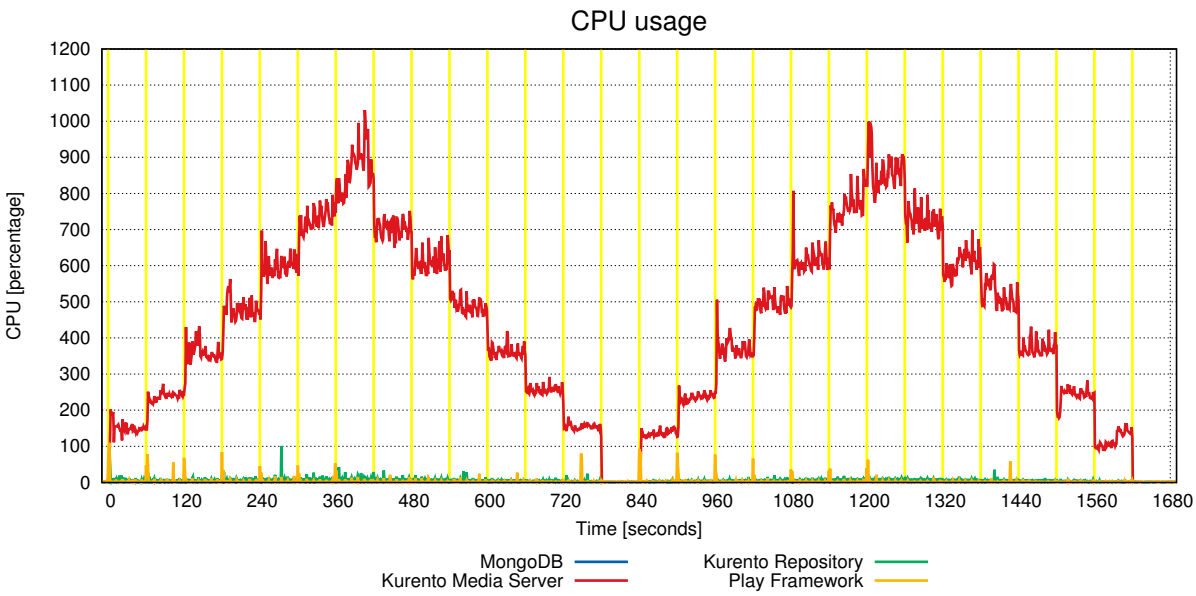


Figure 5.11: Percentage of CPU used during two consecutive test cases

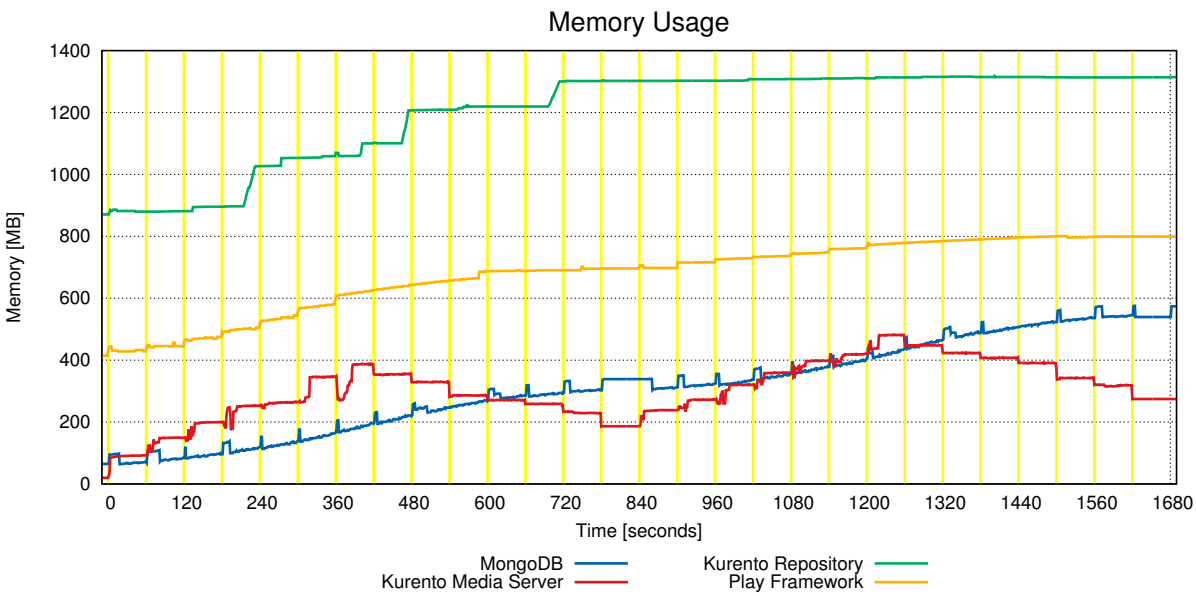


Figure 5.12: Memory usage during two consecutive test cases

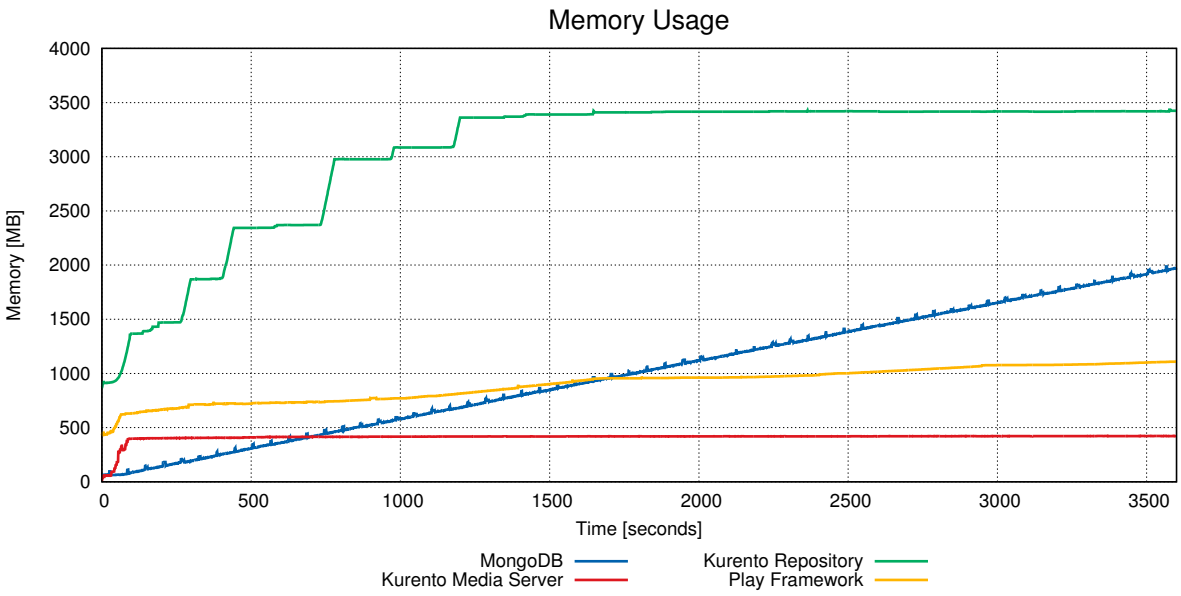


Figure 5.13: Memory usage during one hour with seven clients

2. Create a private conference room, enter and share your screen, add the coordinator to the conference room and chat with him. After that use the collaborative editor in order to write at the same time as the coordinator. Lastly save the editor and leave the conference room.

3. Enter a conference room correspondent as an observer and navigate to the specified annotation. Watch the video until a list of topics is overlayed in the video and choose one of them and leave the conference room.

4. Enter a conference room by sharing their camera and create a time annotation in the instant of time when they entered, navigate to the current time, search for the annotation and leave the conference room.

5. Enter a conference room by sharing their camera and add a subtitle in the video, preview it, specify an interval of time and save it. Then show the provided QR code to the camera and leave the leave the conference room.

The goal of the first task is to evaluating the interface for authentication and friendship management.

The second task is used to evaluate our tools for collaborative content edition.

The goal of the third task is to demonstrate the navigation functionalities and synchronized interactive content superposition.

The fourth task is used to analyse the behavior and difficulties of the user when creating a time annotation.

The goal of the fifth task is to create synchronized overlayed content and present an easier way to synchronize content.

5.4.2 Test Results

In this section we present the results of our usability tests. The first time we tested our solution by providing tasks to users, we have observed that our solution was not perfect. Having faced usability problems during our tests, he had to improve our solution's usability and start the tests again.

First phase

In the first testing phase we have performed tests with just three users and ceased for improvements.

The initial users that tested our solution had difficulties in the first task when finding new friendship requests and adding new friends, as friendship request were only possible to access from the top bar and new user profiles were only accessible through the search bar. In order to solve this problem we have added a menu for adding new users in the friends list and another near the friends list as it can be seen on Figure 5.14.

On the second task one user clicked on the friends list for adding a friend to a room and, as a consequence, left the conference room to the user profile. We solved this problem by showing a pop-up with the user profile and an additional button to add the user to the conference room as it can be seen on Figure 5.15.

On the third task we noticed that users were not expecting to search the content directly from the search bar, instead they used the timeline.

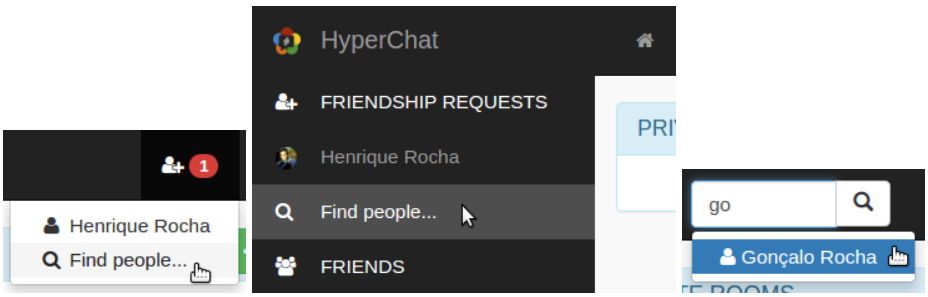
On the fourth task the users did not placed their annotations on right place. We only provide one way to place annotations, which is by dragging them on the timeline. After creating an annotation, users also did not understand that they had to save their changes on the timeline. We solved this problem by saving automatically each time the user changed any annotation. The old user interface can be seen on Figure 5.16.

The three users had success with the fifth task, but they revealed difficulties choosing the time interval for the hyper content. We are aware that this is a difficult task and that's why we have introduced the QR code content creation mechanism which was very easy to learn (with a QR code already printed into paper).

Although content creation through QR codes revealed much easier to perform, this approach implies showing and, as a consequence, recording video with the QR codes, making them visible to other users. In order to solve this problem, we could add a new video sharing mode that allow users to send video to KMS without composing, recording and showing to other users.

We have also gathered comments and suggestions after letting the users explore our system, extra improvements were made such as:

- Giving feedback to user when performing operations such as create superimposed content and save the collaborative editor content.
- Starting with our timeline more zoomed in order to give more sensation of time passing.
- Create content with starting time and duration, instead of ending time.



a) Friend list path b) Friend requests path c) Search bar path

Figure 5.14: Multiple paths for finding people

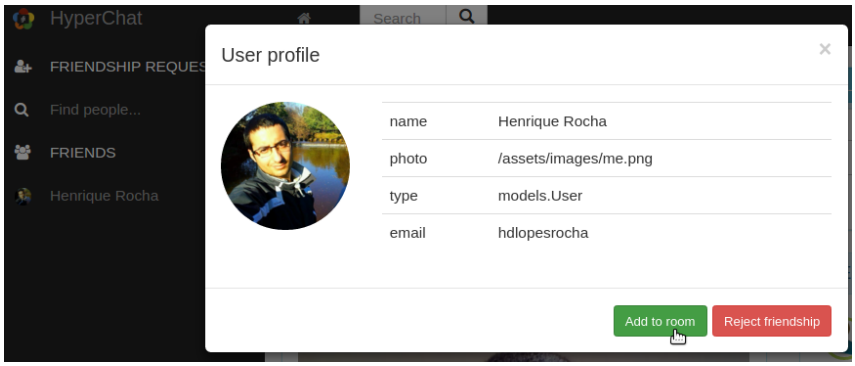


Figure 5.15: Adding user to conference room from user profile

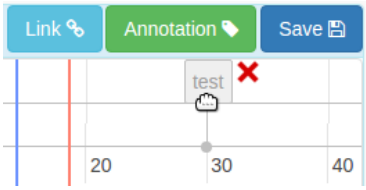


Figure 5.16: Save button that was removed

Second phase

On the second phase of our user interface tests, in a general way, we have noticed great improvements on the learning time.

In order to measure the users learning speed, we have performed tests with experienced users in order to retrieve the optimal task duration and minimal task clicks.

To this end, with regard to optimal task duration and minimal clicks, we obtained the values shown in Table 5.5.

Table 5.5: Metrics for an experienced user

Task	1	2	3	4	5
Duration (seconds)	20	35	30	25	35
Number of clicks	7	10	5	9	7

From the data collected with twenty tests with users, namely the task duration (Figure 5.18), number of clicks (Figure 5.19), number of errors (Figure 5.20), task difficulty (Figure 5.21) and qualitative evaluation (Figure 5.22), we have calculated the confidence intervals in order to understand the most plausible values for each metric.

As a result of both true average and variance being unknown and the usage of a relatively small amount of samples, we had to use *t-distribution* to estimate our metrics confidence intervals.

Accordingly, we have used the following formula to calculate the intervals for the expected value of the true average with 95% confidence:

$$[\bar{x} - F^{-1}_{t_{(n-1)}}(1 - a/2) \times \frac{s}{\sqrt{n}}, \bar{x} + F^{-1}_{t_{(n-1)}}(1 - a/2) \times \frac{s}{\sqrt{n}}] \tag{5.1}$$

The youngest and oldest testers were, receptively, twenty two and thirty eight years old. Figure 5.17 the ages

of the users that tested our system.

Where $a = 0.05$ [9], n is the number of samples (which may not coincide with the number of tests due to failed tasks), $\bar{x}$ is the sample mean, x_i is the sample value and s is the sample standard deviation that is given by:

$$s = \sqrt{\frac{1}{n-1}\Sigma_{i=1}^{n}(x_i - \bar{x})^2} \tag{5.2}$$

According to Figure 5.21, we can observe that most users had less difficulties with the first two tasks, which represents types of tasks that most users are familiar with. As soon as users had to navigate in time, manipulate annotations and create content (respectively *task3*, *task4* and *task5*) we observed that they revealed more difficulties. Most of those difficulties, based on the users feedback, were mainly due to those concepts not being familiar to them.

As we had relatively bad results with some users, we explained to those users that could not conclude the tasks or performed them incorrectly, the most efficient way to perform the requested tasks. Some users suggested to display more hints in order to achieve a faster learning, but afterwards all were impressed and gave us a better evaluation (as seen on Figure 5.22).

Most users gave us worse evaluations on our user interface layout and content editor, which was due to having a lot of tools present in the same web page and some of them being hidden due the screen size. In some cases users had to scroll down in order to find the tools they were looking for.

Another weak aspect was our content editor, which, in fact, we recognize is difficult to work with, mostly due to the amount of information that is necessary to create a synchronized content (starting time, duration and content itself). Some users have suggested that the content should also be present on the timeline so they could be easily dragged and resized (on time).

We are aware that placing content on the timeline will reduce our solutions performance, especially when there is a relatively large amount of content, due to the content that is present on the timeline being loaded all at once. Although, we recognize that for some cases (relatively low amount of content), displaying the content on the timeline could not have a great impact on our solutions performance, we choose not to implement this.

5.5 Chapter Summary

In this chapter we have described how we have deployed our system for allowing a public access and perform tests in a real case environment.

Then, we described the test scenarios that we have created in order to evaluate our system.

The performance tests revealed unexpected results which were crucial to find and solve performance issues. Another important aspect of our performance test results was the understanding of how data flows between our system modules.

In respect to user interface tests, we have obtained valuable feedback that we used to improve our prototype's usability.

In order to understand how good the results were for each performed task, we have defined optimal values for

[9] 95% ($a = 0.05 = 1 - 0.95$) is a common confidence level value used, by researchers, to calculate confidence intervals

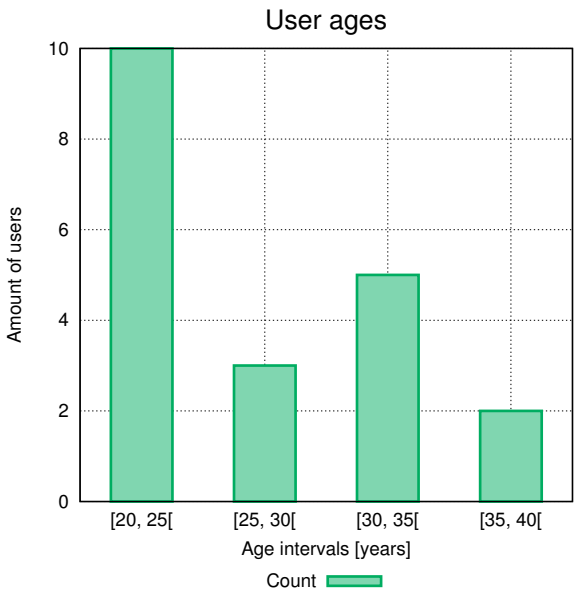


Figure 5.17: Ages of the users that tested our system

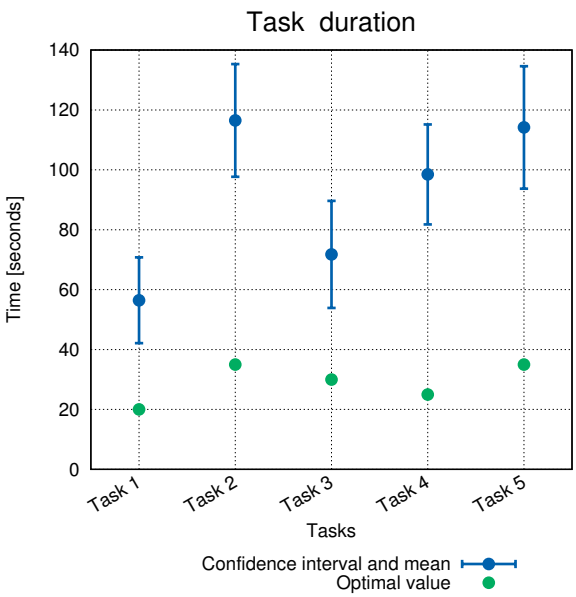


Figure 5.18: Time spent per task

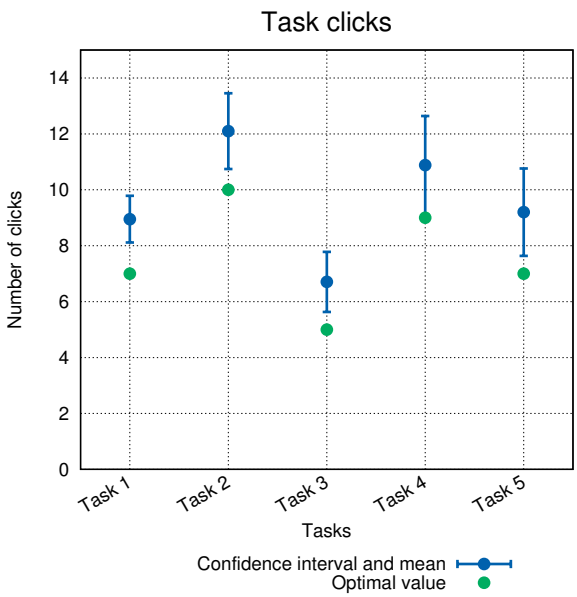


Figure 5.19: Mouse clicks per task

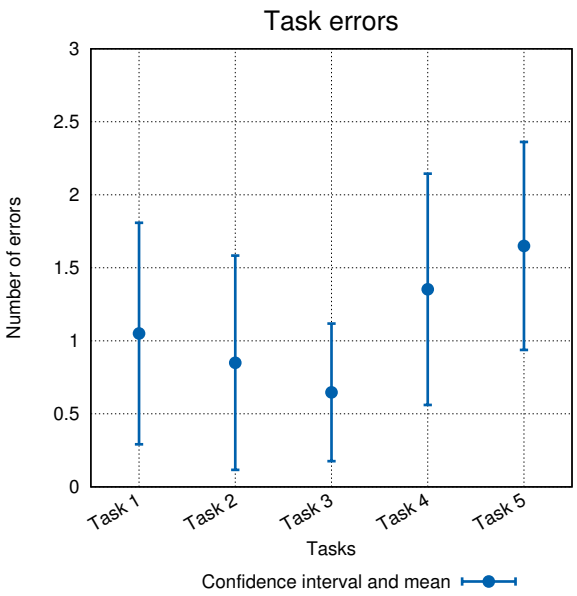


Figure 5.20: Errors per task

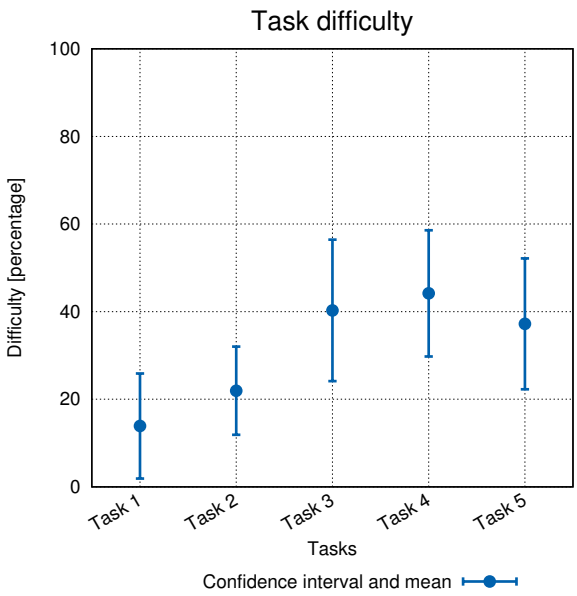


Figure 5.21: Difficulty per task

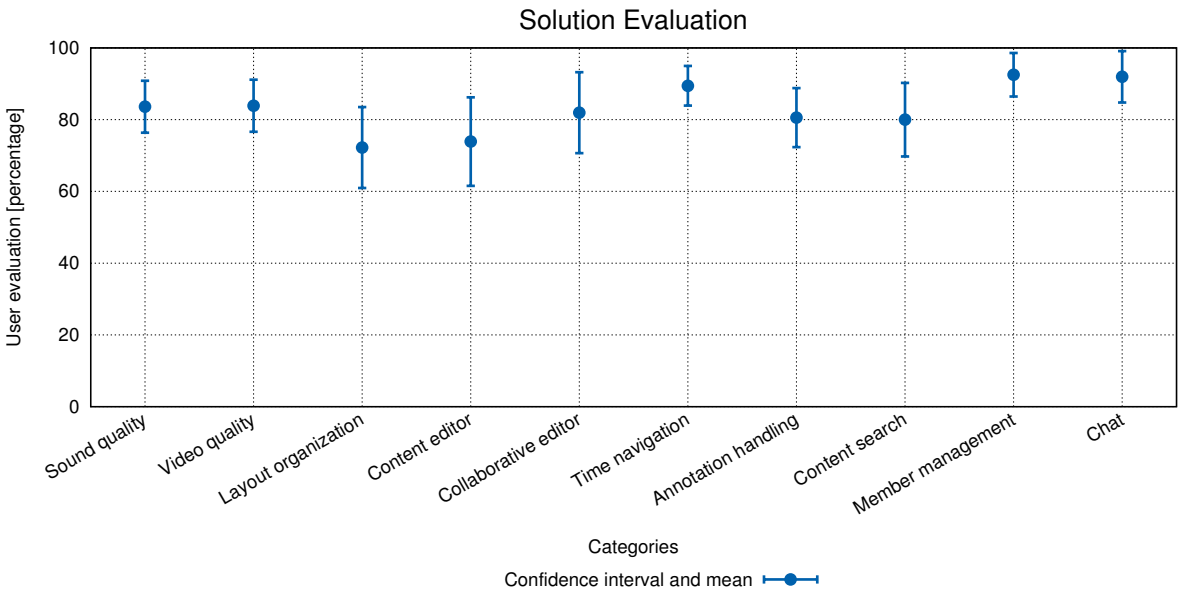


Figure 5.22: Solution evaluation

the different metrics. In general the results reveal a significant difference between observed values and optimal values, which can be explained by the user's lack of experience with with the tools that we provided.

Chapter 6

Conclusions

6.1 Summary

In this thesis, our goal was the development of a web application using *WebRTC* that complements current audio, text and video communications, in order to create rich and collaborative interfaces with the ability to add more content on a future time. Another important goal of this project is the ability to navigate in time by rewinding communications, fast-forward and jump to certain points.

Having this in mind, we started by analyzing the technologies necessary to implement our solution. In a first glance, we have analyzed the problems that real time communication applications face when using the current Internet infrastructure. Next, we analyzed the main components of the *WebRTC* stack. Furthermore, we analyzed how we could implement the signaling component that *WebRTC* does not define. With the transport component analyzed, we then studied how we could expose content in a synchronized and interactive way. We also performed an analysis on the kinds of media types and the particularities of each one when manipulating time. Lastly, on the state of the art context, we analyzed existing frameworks and libraries that could help us implementing a collaborative environment.

Based on our evaluation of the state of the art, we have defined the architecture of our proposed solution.

Although we ended up using our own signaling protocol, we made our first steps using XMPP and we discovered the drawbacks of this approach. For performing communications with XMPP servers, we used *strophe.js* which we found to have a bug that we have fixed.

Another important contribution we have made was a bug fix of a memory leak present on KMS which we discovered when we were doing performance tests with our solution.

We have successfully implemented the basic functionalities of our prototype and spare some time for adding more valuable features such as the ability to create content by exposing *QR codes* to the camera and lastly perform changes to the user interface in order to improve the quality of user's experience.

The performance tests that we have executed showed that our system is stable and more importantly that our web server is lightweight and most of processing power is dedicated to the streaming server.

Our usability tests show results that are considerably worse than the established optimal values due to our solution proposing a different way to communicate that most people are not used to. Although we have obtained

those results, in general our users gave us positive feedback and valuable advices which we have used to improve our system.

6.2 Future Work

Playing back video with a faster rate is not possible using the current version of KMS. This is useful feature that enables users to go through large amounts of video in less time. Even though we expect the availability of that feature in a near future, we have proposed an alternative way to implement faster playback by using *ffmpeg* to convert the video before playing it.

Although we have tested our solution in a powerful machine for the current time, our performance tests revealed that the streaming component uses a lot of resources. We left for a future work a deep analysis on the scalability of our system for which we have proposed different approaches but we have not tested them.

Another aspect we could have tested was the performance of our solution when using TURN servers for relaying the traffic that fails using STUN.

Lastly, we have chosen functionality over security in respect to displaying content to users, which lead to security flaws on our solution. Although we have not solved the security problems, we have proposed a solution that at the same time limits the flexibility of adding new functionalities to our prototype.

Bibliography

[1] Martin, G.: Hypertext to Hypervoice - Linking what we say and what we do. (2012) 6

[2] Paper, A.I.U.W.: Next Generation Internet : IPv4 Address Exhaustion , Mitigation Strategies and Implications for the U. S. (2009) 1–26

[3] Rosenberg, J., Weinberger, J., Huitema, C., Mahy, R.: STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489 (Proposed Standard) (March 2003) Obsoleted by RFC 5389.

[4] Lin, Y.D., Tseng, C.C., Ho, C.Y., Wu, Y.H.: How nat-compatible are voip applications? Communications Magazine, IEEE **48**(12) (December 2010) 58–65

[5] Rescorla, E., Modadugu, N.: Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard) (January 2012) Updated by RFC 7507.

[6] Handley, M., Jacobson, V., Perkins, C.: SDP: Session Description Protocol. RFC 4566 (Proposed Standard) (July 2006)

[7] Fette, I., Melnikov, A.: The WebSocket Protocol. RFC 6455 (Proposed Standard) (December 2011)

[8] Paterson, I., Smith, D., Saint-Andre, P., Moffitt, J., Stout, L., Tilanus, W.: Bidirectional-streams Over Synchronous HTTP (BOSH). XEP-0124 (Draft) (April 2014)

[9] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., Schooler, E.: SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard) (June 2002) Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141.

[10] Lonnfors, M., Costa-Requena, J., Leppanen, E., Khartabil, H.: Session Initiation Protocol (SIP) Extension for Partial Notification of Presence Information. RFC 5263 (Proposed Standard) (September 2008)

[11] Campbell, B., Rosenberg, J., Schulzrinne, H., Huitema, C., Gurle, D.: Session Initiation Protocol (SIP) Extension for Instant Messaging. RFC 3428 (Proposed Standard) (December 2002)

[12] Muldowney, T., Miller, M., Eatmon, R., Saint-Andre, P.: SI File Transfer. XEP-0096 (Draft) (April 2004)

[13] Saint-Andre, P.: Multi-User Chat. XEP-0045 (Draft) (February 2012)

[14] Paterson, I., Saint-Andre, P., Stout, L., Tilanus, W.: XMPP Over BOSH. XEP-0206 (Draft) (April 2014)

[15] Stout, L., Moffitt, J., Cestari, E.: An Extensible Messaging and Presence Protocol (XMPP) Subprotocol for WebSocket. IETF RFC 7395 (October 2015)

[16] Eatmon, R., Hildebrand, J., Miller, J., Muldowney, T., Saint-Andre, P.: XEP-0004: Data Forms. (2007)

[17] Saint-Andre, P.: Extensible Messaging and Presence Protocol (XMPP): Core. RFC 3920 (Proposed Standard) (October 2004) Obsoleted by RFC 6120, updated by RFC 6122.

[18] Chainho, P., Haensge, K., Druesedow, S., Maruschke, M.: Signalling-on-the-fly: Sigofly. In: Intelligence in Next Generation Networks (ICIN), 2015 18th International Conference on. (Feb 2015) 1–8

[19] Sampaio Gradvohl, A.L., Iano, Y.: Matching interactive tv and hypervideo. Latin America Transactions, IEEE (Revista IEEE America Latina) **5**(8) (Dec 2007) 579–584

[20] Sawhney, N., Balcom, D., Smith, I.: Authoring and navigating video in space and time. MultiMedia, IEEE **4**(4) (Oct 1997) 30–39

[21] Shipman, F., Girgensohn, A., Wilcox, L.: Creating navigable multi-level video summaries. In: Multimedia and Expo, 2003. ICME '03. Proceedings. 2003 International Conference on. Volume 2. (July 2003) II–753–6 vol.2

[22] Zhou, T., Jin, J.: A structured document model for authoring video-based hypermedia. In: Multimedia Modelling Conference, 2005. MMM 2005. Proceedings of the 11th International. (Jan 2005) 421–426

[23] Bulterman, D.: Smil 2.0 part 1: overview, concepts, and structure. MultiMedia, IEEE **8**(4) (Oct 2001) 82–88

[24] Robie, J., Nicol, G., Wood, L., Wilson, C., Champion, M., Isaacson, S., Byrne, S.B., Jacobs, I., Sutor, R.S., Hors, A.L.: Document object model (DOM) level 1. W3C recommendation, W3C (October 1998) http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001.

[25] Bulterman, D.C.A., Jansen, A.J., Cesar Garcia, P.S.: Video On The Web: Experiences From SMIL And From The Ambulant Annotator. In Le Hégaret, P., ed.: Collected Position Papers, W3C Video on the Web Workshop, W3C (December 2007) –

[26] Bulterman, D., Jansen, J., Cesar, P., Mullender, S., DeMeglio, M., Quint, J., Vuorimaa, P., Cruz-Lara, S., Kawamura, H., Weck, D., Hyche, E., Pañeda, X.G., Melendi, D., Michel, T., Zucker, D.F.: Synchronized multimedia integration language (smil 3.0). World Wide Web Consortium, Recommendation REC-SMIL3-20081201 (December 2008)

[27] Gaggi, O., Danese, L.: A smil player for any web browser. In: DMS, Knowledge Systems Institute (2011) 114–119

[28] McAlarney, J., Haddad, R., McGarry, M.P.: Modeling network protocol overhead for video. In: Computer Modeling and Simulation (EMS), 2010 Fourth UKSim European Symposium on. (Nov 2010) 375–380

[29] Schulzrinne, H., Casner, S., Frederick, R., Jacobson, V.: RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard) (July 2003) Updated by RFCs 5506, 5761, 6051, 6222.

[30] Schulzrinne, H., Casner, S.: RTP Profile for Audio and Video Conferences with Minimal Control. RFC 3551 (Standard) (July 2003) Updated by RFC 5761.

[31] Ivov, E., Marinov, L., Hancke, P.: COnferences with LIghtweight BRIdging (COLIBRI). XEP-0340 (Experimental) (January 2014)

[32] Mauve, M., Hilt, V., Kuhmunch, C., Effelsberg, W.: A general framework and communication protocol for the transmission of interactive media with real-time characteristics. In: Multimedia Computing and Systems, 1999. IEEE International Conference on. Volume 2. (Jul 1999) 641–646 vol.2

[33] Hilt, V., Mauve, M., Kuhmünch, C., Effelsberg, W.: A generic scheme for the recording of interactive media streams. In Diaz, M., Owezarski, P., Sénac, P., eds.: Interactive Distributed Multimedia Systems and Telecommunication Services. Volume 1718 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1999) 291–304

[34] Nichols, D.A., Curtis, P., Dixon, M., Lamping, J.: High-latency, low-bandwidth windowing in the jupiter collaboration system. In: Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology. UIST '95, New York, NY, USA, ACM (1995) 111–120

[35] Morris, R., Thompson, K.: Password security: A case history. Communications of the ACM **22**(11) (1979) 594–597

[36] Schütt, K., Kloft, M., Bikadorov, A., Rieck, K.: Early detection of malicious behavior in javascript code. In: Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence. AISec '12, New York, NY, USA, ACM (2012) 15–24