BACHELOR'S THESIS

TITLE: Remote Interactive Whiteboard Web Application with Synchronized Video Playback

DEGREE: Bachelor's degree in Telecommunication Systems

AUTHOR: Montse Guiu Canteras

DIRECTOR UPC: Eduard García Villegas

DIRECTOR: Héctor Cano Vázquez

DATE: September, 23rd of 2015

Títol: Aplicació web de reproducció de vídeo sincronitzat i pissarra remota

Autor: Montse Guiu Canteras

Director UPC: Eduard Garcia Villegas

Director extern: Héctor Cano Vázquez

Data: 23 de setembre del 2015

Resum

Aquesta tesi aborda la necessitat de poder revisar els entrenaments fets en centres d'alt rendiment esportiu i fer-ne l'anàlisi tàctic, a partir del visionat de les gravacions corresponents, quan un o varis dels convocats no hi pot assistir presencialment.

La solució que proposem és una aplicació web basada en un producte ja existent, anomenat *Command Center Sports*, amb l'objectiu d'integrar-la dins d'aquest i millorar-lo afegint una pissarra remota construïda amb l'element canvas de HTML5, la qual es sincronitza utilitzant el protocol WebRTC. Això proporciona un marc bidireccional en el qual els usuaris podran participar remotament a les sessions d'entrenament amb la resta d'assistents. L'aplicació no requereix instal·lar connectors, controladors o extensions i funciona de forma nativa en un navegador. Per tant, pot funcionar tant en ordinadors, tauletes i telèfons intel·ligents, independentment del sistema operatiu.

El projecte ha cobert totes les fases del cicle de desenvolupament de productes, incloent: la recopilació dels requeriments del client, una recerca exhaustiva de l'estat de l'art, el disseny de l'arquitectura utilitzant diagrames UML, la execució de l'aplicació, anomenada Remote Instant Video Feedback, i proves amb el client i anàlisi dels seus comentaris, tant durant el desenvolupament del prototip com després de la seva presentació, per tal de poder ajustar la aplicació a les seves necessitats i incloure noves millores.

Aquest document detalla els passos realitzats fins arribar a la solució final, i on es documenten els coneixements i les tecnologies que s'han estudiat, però no pot no pot captar adequadament el creixement personal i professional que ha representat per mi, o el profund agraïment cap als meus dos directors.

Title: Remote Interactive Whiteboard Web Application with Synchronized Video Playback

Author: Montse Guiu Canteras

Director UPC: Eduard García Villegas

Director extern: Héctor Cano Vázquez

Date: September 23rd 2015

Overview

This thesis addresses the need for elite sport training centers to review exercises and discuss tactics based on captured sport footage *when some of the attendees are not present*.

The proposed solution is a Web App that builds on top of the existing Sports Command Center application and enhances it with an HTML5 Canvas based whiteboard, which is synchronized using the WebRTC protocol. This provides a bidirectional, fast and reliable framework on which attendees can draw and express their thoughts, tactics and comments during briefing and debriefing sessions. The application requires no installation of plugins, drivers or extensions and works natively in the browser. It therefore runs in desktop PCs, tablets and mobile phones regardless of the Operating System.

The project has covered all phases of the product development cycle, including requirement gathering, intensive research of the state of the art, design of the architecture with the help of UML diagrams, implementation of the application, dubbed *Instant Video Feedback*, and testing with customers, including looping in customer feedback during development and upon release to further tailor and improve the application.

This document details the steps taken towards the finished solution, and documents the lessons and technologies learned, but cannot adequately capture the personal growth it has represented, or my deep gratitude to my supervisors and our customers.

I would like to express my gratitude and sincere appreciation to my thesis directors Héctor Cano and Eduard Garcia for giving me the chance to carry out this project. Their guidance and encouragement through the course of this work has been very useful and a valuable aid to perform this thesis.

Also, I dedicate this project to my friends Miquel and Oscar for their continued support both personal and professional.

Table of Contents

Table of figures

Introduction

Technology has become pervasive, redefining how people interact with the Internet and its contents. New devices and applications crop up every day, putting more and more information and control at our fingertips. And yet it is about so much more than information: the internet connects us, and real-time, quick response, reliable communication has become the norm.

It is therefore not surprising that teleconference and screen sharing is now common place in so many professional and non-professional environments: troubleshooting, on-line presentations, collective on-line work sessions and gaming, are just some of the examples.

In this line, this project aims to fill an identified gap in the field of elite sports. In this area, the use of supporting technologies to review gameplays and illustrate strategies is growing to be a basic necessity, and this is even truer as the professional level and budget increase. And yet, the tools available still leave much to be desired. In this project we extend an existing application, Sports Command Center, to support remote collaboration. While numerous applications exist that cover some of the use cases, there isn't yet one that is tailored to the needs of sports professionals and assists in remote training sessions.

The Sports Command Center application is currently already in use in training facilities, and includes a system of IP cameras, a server for video stream recording and playback and a NAS to store the video streams. The system offers two interfaces: an application running in the server and a user facing web application to display the videos. The work done in this project further extends the client facing up with new remote interface: the Instant Video Feedback Screen. Through it, the coach can attend and interact with those present at the training session, as well as review recorded sessions for tactical analysis. The need arose from customer requests to "enable the technical staff and the athletes to conduct on-line training sessions in the sport team's facilities even if the coach is not present". Upon review by customers, we are proud that the finished project is to the customer's liking, who now sees further use cases and application areas within their training workflows.

Fulfilling the customer needs has a good understanding of the existing technologies, and detailed understanding of the requirements involved. For example, the application had to work with as little prior installation as possible, to be flexible in where it was used, as well as support multiple platforms and interaction modes, from a PC to a tablet. The application needed to provide video playback synchronization, and still support drawing tactical analysis graphs on it in a bidirectional manner, so that both peers could interact. The application needed to "feel agile and light" (i.e. respond quickly and have low latency) and expressive enough for the needs of the task.

The following are a high-level overview of what the project aimed to solve:

- Overlay a drawing surface on top of a synchronized video playback
- Provide bi-directional synchronized drawing capabilities to both communication peers
- Ensure low latency and quick response in the user experience
- The communication must be both reliable and secure
- The application must integrate within the existing framework of Sports Command Center

In addition, both the project and the application are committed to using the newest technologies available and open standards when possible, in order to stay ahead of the market. It is also a deep seated value that reinventing the wheel serves no purpose, and we have therefore put a lot of emphasis on the evaluation of potential solutions, which has provided the author with a unique opportunity to vastly deepen the knowledge of the area as well as growing as a professional.

The approach in this project has therefore been carefully chosen, and roughly matches the sections in this document. We started off with analysis of the state of the art, as presented in Section 1. The goal was to evaluate and learn different technologies and, once understood, match what we learned against the requirements of the project and the expectation of the customers.

It is through this that the choice was made for using a Web Application that injects an HTML 5 Canvas that's synchronized using the WebRTC protocol. The details of the design were illustrated using UML diagrams in order to clarify the solution as the project advanced. This is covered in detail in section 2.

Once the application was finished an iterative process started where customer feedback was taken into account to better the application. There were a few rounds of this where the usability and performance of the application was improved, and we describe this in section 3.

Instant Video Feedback is now complete and, while there are improvements to be made in future releases, some of which is covered in the Next Steps section, we are satisfied about the very positive customer feedback. The application fulfills their needs, and there are already requests for future improvements.

This has been a learning experience and a unique opportunity for growth, for which I am thankful for the support and patience of both my supervisors and our customers.

CHAPTER 1. State of the art and design choices

Our objective is to add remote features to an existing web app. In the next sections, different technical approaches are discussed with the aim of choosing what suits our proposal best, focusing on the analysis of application framework, how caller and callee interact and, finally, how to introduce a whiteboarding tool.

1.1. Application Framework

1.1.1. Requirements

One of the main improvements we want to add to the existing product is giving the clients the possibility to connect to their video training system from anywhere and from any device. Therefore, our solution has to be able to run in any existing platform and equipment, regardless of the technology they use. In other words, we need to develop a “zero installation required” and multiplatform solution.

1.1.2. State of the art

An application is a kit of computer programs created to allow the user to do a concrete task or activity. We can distinguish between native, web and hybrid applications.

Native applications are designed specifically for a given platform or device. Their developers need to implement different solutions of the same product depending on the operating system or if it is going to be installed in a computer, a tablet or a smartphone. This type of applications have multiple benefits such as: access to the local resources of the device (local storage, GPS, camera, etc), work offline, and run faster among others. On the other hand, due to the multiple platforms and types of devices, using native applications implies an important development cost because of the multiple versions of the same application the developer needs to build and maintain if he wants it to be compatible with all the platforms available on the market. The same happens with the maintenance and support of these applications.

An alternative approach are Web applications, where an application program is stored on a remote server and its user interface is delivered over the Web through a browser each time it is run (unless it’s cached), using technologies such as HTML5, CSS or JavaScript. The proliferation of standards is what makes web applications accessible from any browser, regardless of the computer

architecture or operating system we are working with. Thus, they are described as cross-platform or platform agnostic and because of that, there are significant savings in development and maintenance costs. However, an Internet connection is needed to access the content. Since they run in a web browser, the user experience and presentation may sometimes be inferior compared to native applications, but this may not be an issue anymore after the final release of HTML5 in 2014. This markup language adds several new syntactic features like the video, audio and canvas elements along with Scalable Vector Graphics (SVG) content and MathML for mathematical formulas. The requirement for applications to be online are also slowly fading with offline in-browser support, but at any rate, this is not an issue for applications that require on-line content in any case.

However, the two are not mutually exclusive because many applications contain elements of both native and Web applications. Programs that combine the two approaches are sometimes referred to as hybrid applications.

1.1.3. Choice

As we said earlier, our aim is to create a cross-platform application able to play videos and draw on it. After consider the main features of each type described above we will develop a web application using HTML5 combined with JavaScript and other libraries that will be highlighted in the following sections.

1.2. Screen sharing and Communication channel

1.2.1. Requirement

After deciding which type of application we will design, there are other features that our solution needs to fulfill. It has to be reliable, which, in real-time development, means ensuring the delivery of critical data to guarantee the communication between both sides and the screen sharing utility. It also has to use a secure channel to prevent users from eavesdropping or modifying on the peer to peer connection. And finally, our proposal has to be able to work through the majority of connection setups (multi homing, firewalls and NATs).

1.2.2. State of the art

Nowadays there are many applications for screen sharing on the Internet: for technical support, to work on line, on-line presentations, etc. In other words, this type of utilities allow for several people to watch and even modify one same process.

Most of these many options implement the feature of streaming a video, but the participants need to install a suitable client on the used platform, as is the case with Skype or Team Viewer, to add a plugin to the browser or install additional software as does Screanleap, which requires Java. In the last years, the progress in that area has moved forward and there are some applications that do not even need any extra installation and require only an account, as with Hangouts, which uses Web Real-Time Communication (WebRTC). WebRTC turns real-time communication into a standard feature that any web application can use with a simple JavaScript API.

The WebRTC project is supported by Google, Mozilla and Opera amongst others and was first released in 2011. It is a collection of standards, protocols and JavaScript API's that combined allow peer-to-peer audio, video and data sharing between browsers or peers without using proprietary software or third-party plugins. To provide these features, a lot of new functionality in the browser is required. WebRTC developers have hidden all this complexity behind three main API's: **MediaStream**, **RTCPeerConnection** and **RTCDataChannel**. [1]

- **MediaStream** (aka *getUserMedia*): this interface gets access and allows the manipulation of streams of multimedia data (audio and/or video)
- **RTCPeerConnection** (aka *PeerConnection*): sets up an audio/video call between the caller and the callee. It also includes tools for bandwidth management and encryption.
- **RTCDataChannel** (aka *DataChannels*): this API allows browsers to share generic data via peer-to-peer.

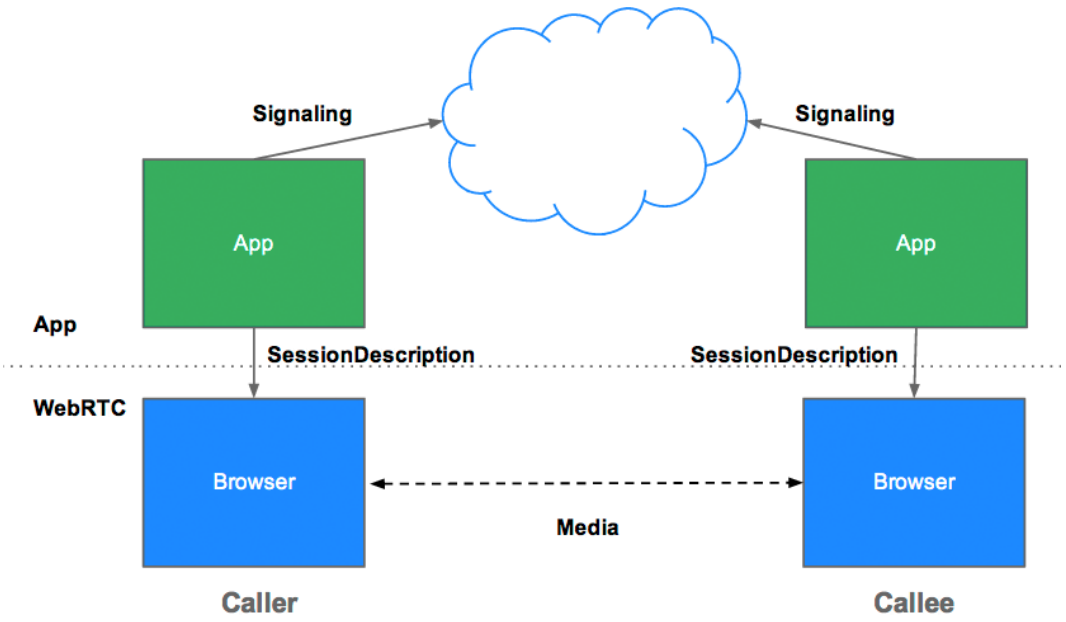

Fig. 1.1 JSEP Architecture [1]

Before peer-to-peer streaming can begin, the caller and the callee need to learn and exchange local and remote media information (resolution, codec capabilities,

etc.) and network information (IP, ports, etc.). All this information is embedded in *RTCSessionDescription* objects, exchanged by means of Session Description Protocol (SDP) messages. The exchange of these objects proceeds by exchanging an offer and an answer between peers. This offer/answer architecture is called JSEP (JavaScript Session Establishment Protocol). Fig. 1.1 shows JSEP architecture and helps to understand the process of signaling and streaming.

The WebRTC standard does not define a specific channel to perform the signaling, so any protocol or mechanism can be used to exchange the SDP objects. Signaling is not part of the RTCPeerConnection API.

WebRTC apps can use the ICE (Interactive Connectivity Establishment) framework to overcome the complexities of real-world networking. To enable this to happen, the application must pass ICE server URLs to RTCPeerConnection, as described below.

ICE tries to find the best path to connect peers. It tries all possibilities in parallel and chooses the most efficient option that works. ICE first tries to make a connection using the host address obtained from a device's operating system and network card; if that fails (which it will for devices behind NATs) ICE obtains an external address using a STUN server, and if that fails, traffic is routed via a TURN relay server. [2]

In other words:

- A STUN server is used to get an external network address.
- TURN servers are used to relay traffic if direct (peer to peer) connection fails.

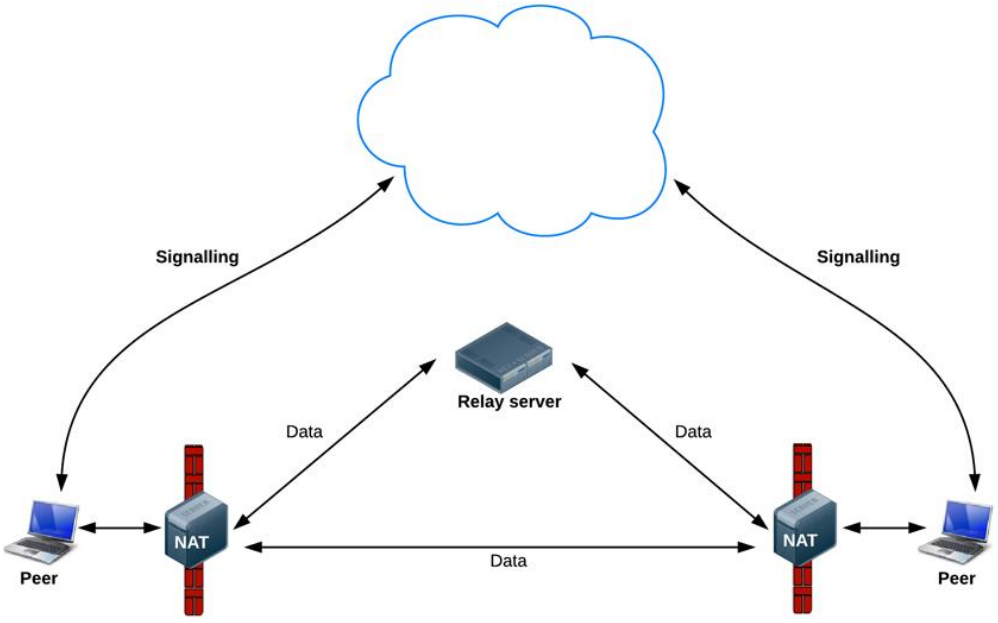

Fig. 1.2 WebRTC data pathways

Once caller and callee have discovered each other and exchanged information about their capabilities, a call session is initiated, and real-time data communication can begin.

The RTCDataChannel API [3]enables peer-to-peer exchange of arbitrary data. There are many potential use cases for the API, including:

- Gaming
- Remote desktop applications
- Real-time text chat
- File transfer
- Decentralized networks

The API has several features to make the most of RTCPeerConnection and enable powerful and flexible peer-to-peer communication:

- Leveraging of RTCPeerConnection session setup.
- Multiple simultaneous channels, with prioritization.
- Reliable and unreliable delivery semantics.
- Built-in security (DTLS) and congestion control.
- Ability to use with or without audio or video.

Communication occurs directly between browsers, so RTCDataChannel can be much faster than WebSocket even if a relay (TURN) server is required

Finally, WebRTC has several features to avoid security problems, such as the use of secure protocols like DTLS, all WebRTC components are encrypted and as they run in the browser sandbox, do not need installation or updating to avoid access to malware of viruses.

1.2.3. Choice

To achieve the development of our application: a reliable, secure real-time web application able to establish connections across different types of networks, we have chosen to implement it using the recently developed widespread WebRTC Project.

As we exposed in previous section, the WebRTC standard does not define a specific channel to perform the signaling, in this project we will use Websocket and JSON (described in section 2.4.2.3).

1.3. Whiteboarding

1.3.1. Requirements

The application calls for a remote, interactive, highly expressive and extensible surface upon which peers can draw bidirectionally in order to annotate training sessions.

1.3.2. State of the art

HTML5 Canvas [4]started off as the top contender for this part of the application. Given the need to work within a web application, the only real alternatives where Adobe Flash and SVG.
Adobe Flash has received very bad press on account of the security flaws discovered recently, and has been ramping down for years in favor of HTML 5 technologies. We therefore consider it to be an application on the way to deprecation and is certainly not a future proof tech.

SVG (Scalar Vector Graphics) [5] provides modern support for drawing in the browser, but it mostly centers around geometric shapes (vectors), where our customers were rather targeting the expressiveness of free drawing, even just using a fingertip.

HTML5 Canvas on the other hand provides a modern alternative to share bitmap graphics, and is nowadays well supported in all mainstream browsers.

1.3.3. Choice

Given the customer needs and the evaluation above, HTML 5 Canvas was the only viable option and our solution of choice.

CHAPTER 2. Design and Implementation

2.1. Remote Instant Video Feedback

We want to implement some improvements to an existing web application that already allows annotating video streams provided from a server. This application is called Remote Instant Video Feedback (R-IVF).

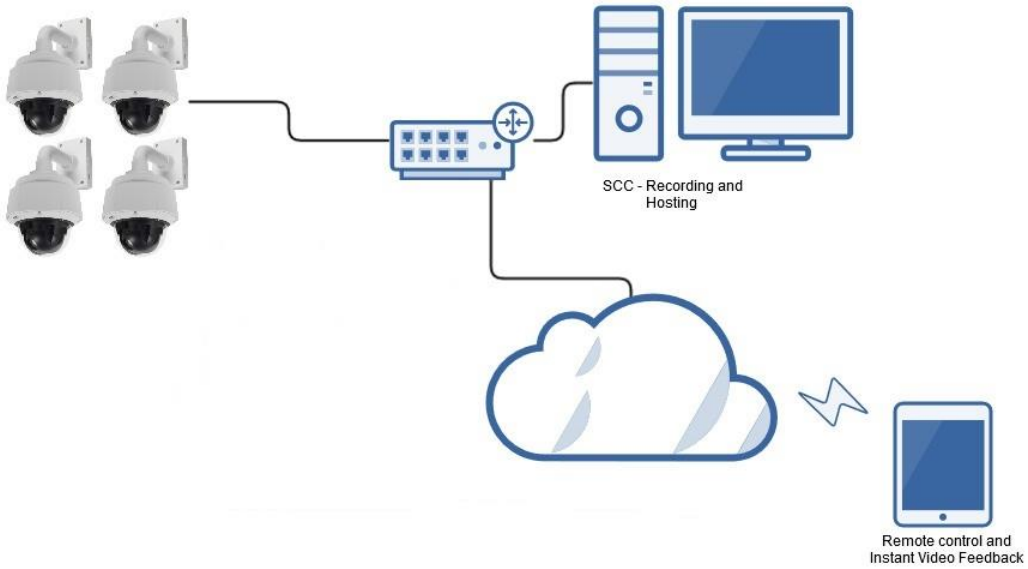

Fig. 2.1 IVF architecture

The system is comprised by:

- a set of IP cameras who record the training session,
- a workstation which works as a web and video server,
- a NAS (Network Attached Storage) where the video streams are stored,
- the web application IVF that allows the trainer or the technical staff to analyze the training sessions from any browser.

In the Fig. 2.2, we can see IVF's screenshot. There are several functionalities implemented:

- Several buttons that allow the user to move backwards and forwards in the video timeline in steps of 1, 5, 15 or 30 seconds. Also, the user can add bookmarks (BM button) and list them for a later review.
- The video and the canvas elements have the same dimensions and they overlap onto the same area in the browser, enabling the technical staff to

draw diagrams and graphs over the video and perform the tactical analysis.

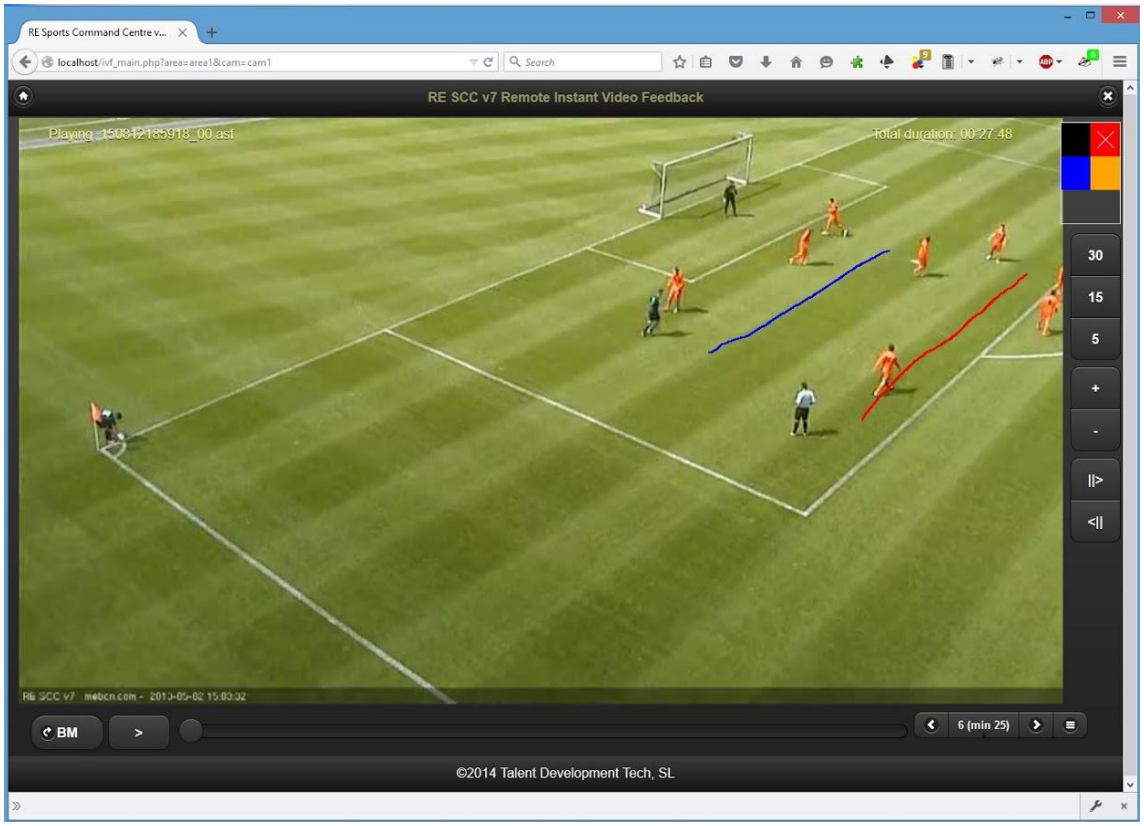

Fig. 2.2 Screen shoot of IVF web application

This application can be a helpful utility in the posterior tactical analysis of any type of sports. However, such application does not allow the on-site team to participate in the analysis in real-time. That is, we detected the need of having teams cooperating remotely to review strategies and outline plays.

Although you can find in the Internet many solutions for screen sharing or online working utilities, none of these existing solutions allow video playback and annotation in real-time.

Both functionalities are oriented to the world of sports, due its nature, there was the need for remote collaboration with zero installation to enable cooperation at any time, in any place, with broad platform support and connectivity in most network scenarios.

Hence, as outlined in state of the art, the best suited solution was a web based client using open standards like WebRTC and HTML5, which greatly facilitate this task. In Fig. 2.3 is described the final system architecture once our application is added.

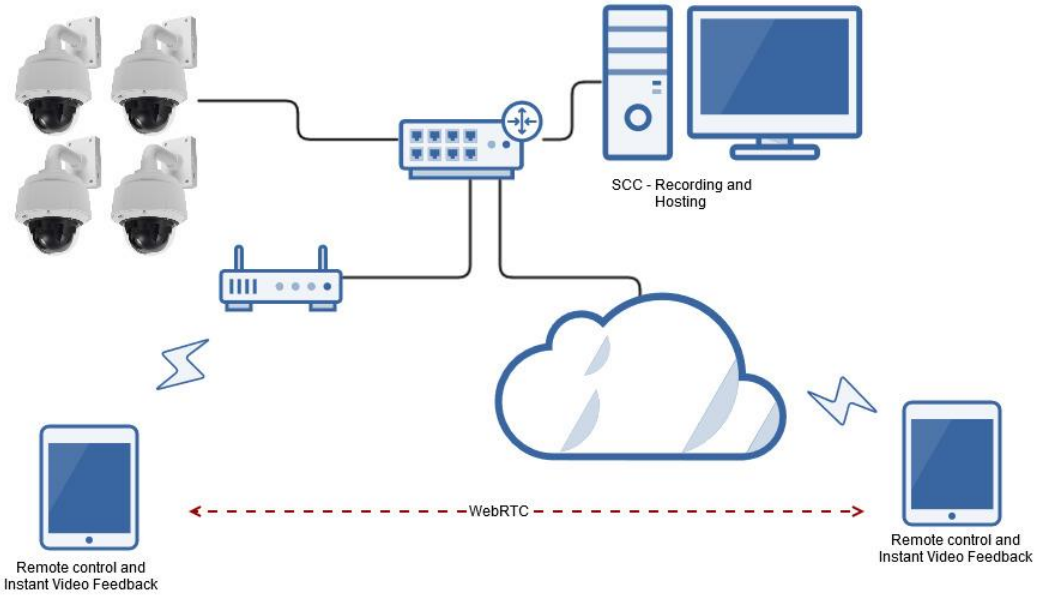

Fig. 2.3 IVF architecture

2.2. Use Case diagrams

The following diagram outlines the main use cases [6] for our application. They can be divided in three main blocks:

1. Those that relate to establishing and managing interaction sessions:

    - Initiate WebRTC Session: peer to peer communication established.
    - Terminate Session: ends the session between the two browsers.

2. This cases is related to playback a given video and synchronization across clients:

    - Synchronize video stream: reproduce simultaneously a video stream in both browsers.

3. The functionality that provides remote interactive whiteboard and chat services:

    - Draw Whiteboard Content: synchronized drawing in both browsers.
    - Chat Content: chat interface.

Note that, once the communication is established, both users, caller and callee, are fully equivalent to interact with the canvas and chat elements, but only one of them is able to initiate the communication.

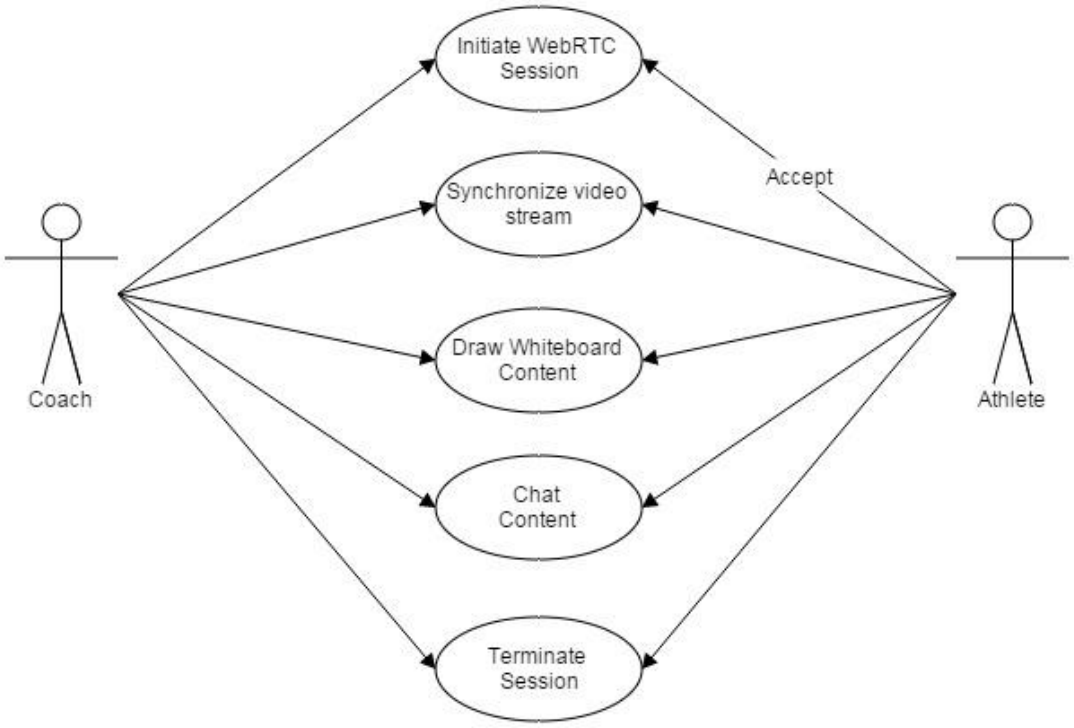

Fig. 2.4 Use Case Diagram

2.3. Architecture

This system requires no additional installation and, due to the wide range of supported platforms, have resulted in a heavily client biased architecture with server support for video storage, recording and management.

The clients run standards-compliant browsers, such as Chrome, FireFox, Safari, Internet Explorer and Edge, to ensure interoperability. The application runs locally in the browser within the JavaScript sandbox to avoid that malicious software can access to the local file system.

Clients communicate to two main end points: directly with other clients using peer to peer fashion, and to the backend for video browsing and downloading. All synchronization tasks both for video and whiteboarding are done without server support, once the session between both clients is established, we pass the instructions for video and canvas elements as is shown in Fig. 2.5 which represents the Component Diagram. [7]

It is especially important to note that:

- Because of the heavy use of WebRTC protocols, connections surpass most firewalls and corporate proxies.

- Rather than reinventing the wheel, the hard problems of synchronizing clocks, guaranteeing endpoint health and connection maintenance and throttling is handled using well known open standards.

2.3.1. Component Diagram

The following diagram shows the overall architecture:

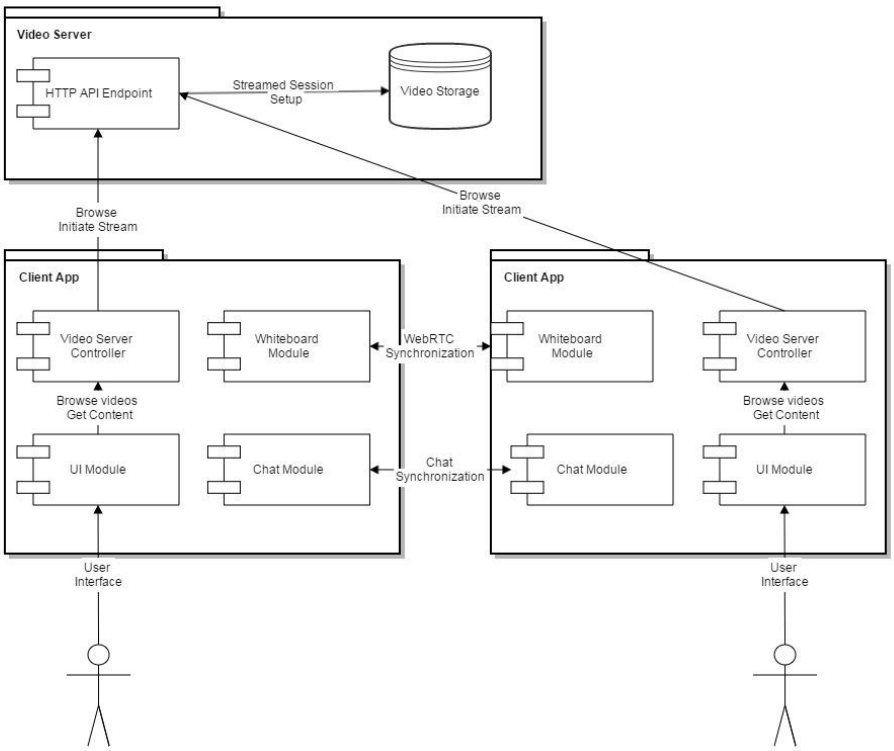

Fig. 2.5 Component Diagram

2.3.2. Sequence diagrams

Because the communication endpoints are always run on single-threaded JavaScript, all calls to external services are asynchronous, based either on AJAX requests or Web Sockets. This, in turn, guarantees that the rest of the UI will always remain usable and will respond quickly.

Where communication occurs over a long period of time with multiple events, we favored using the WebRTC channels, which try to maintain a single open connection, therefore saving the high cost of renegotiating new connections at every request, and greatly improving response time.

The main sequence diagrams are illustrated below. Fig 2.7 depicts how the different objects interact to carry out the Whiteboard instance and the same for Fig. 2.8 for the Video Instance.

It is relevant to note that:

- Both users need to log into the system through their browsers.
- IVF application always serves the last recorded training session.
- The synchronization is performed sending the commands through the WebRTC channel. For Whiteboard, we send drawing commands and coordinates of the canvas. For Video playback, we send playing commands and timeline positions.
- In this scenario, both users, trainer and athletes, can draw and playback the video. But only the trainer can offer to start the WebRTC session.

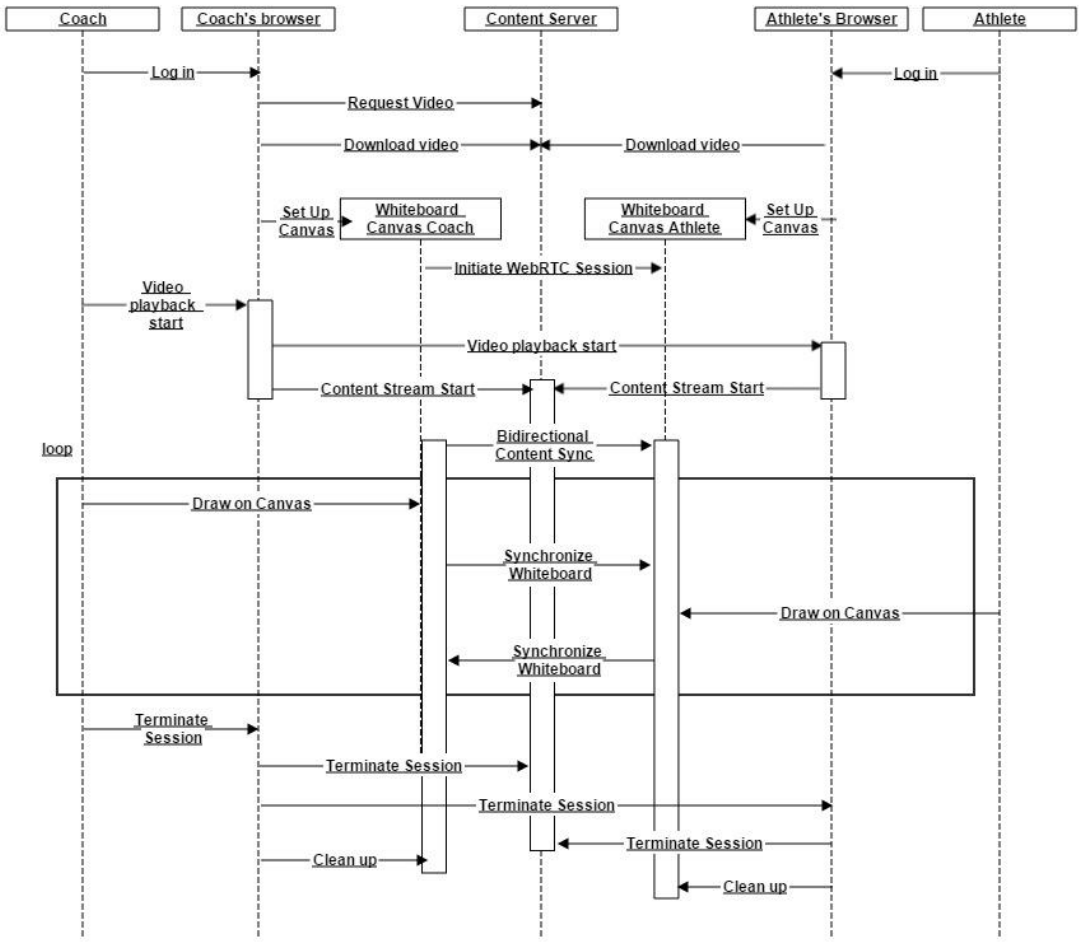

Fig. 2.6 Whiteboard Sequence Diagram

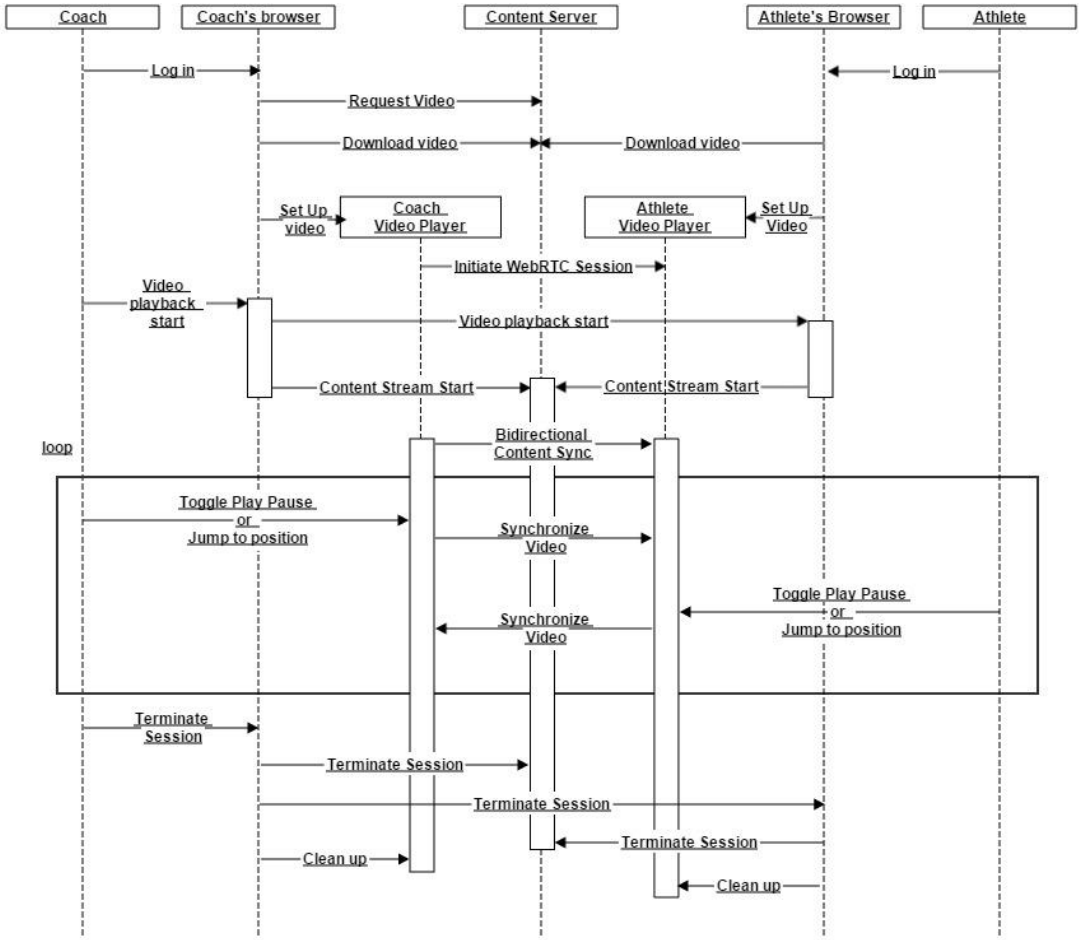

Fig. 2.7 Video Sequence Diagram

2.3.3. Activity diagrams

The actual flow of the application is fairly straightforward as shown in the activity diagram shown in Fig 2.9. This depicts roughly the flow for a single session as long as it is active, marked as a loop.

We can differentiate two main parts: first one for the signaling and WebRTC peer-to-peer connection. And the second part, after WebRTC Data Channel is open.

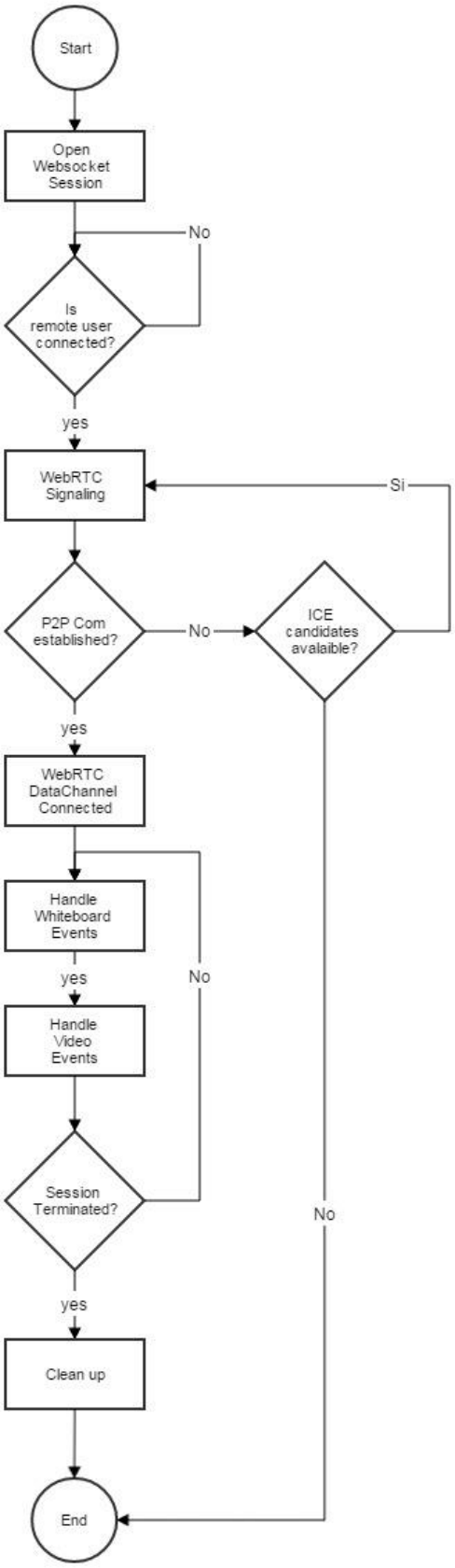

Fig. 2.8 Activity Diagram

2.4. Detailed design and development choices

The development was done on a Linux virtual machine over a windows host. Libraries and development requirements were installed on the virtual machine. Version control and code editing was done on the host machine, and tested on the virtual machine.

2.4.1. Development Environment

To carry out the development and tests a virtual Ubuntu machine was used. This way the host computer could be kept clean and the development environment could be reseted and reproduced if needed. Also it allowed to monitor the impact the running application had on a simple server.

A very simple Ubuntu server, with a single processor and just 512MB of RAM.

2.4.1.1. Vagrant and Virtualbox

Vagrant [8] is a tool for building complete development environments. As a virtualization software, allows the creation of repeatable virtual environments on top of virtual machines. It is written in Ruby but it can be used for other programming languages.

The deployment of a web application is built by containers for the entire server stack and it does not require other software installed on the virtual machine.

The usage of containers allows building up a web server combining different technologies and allowing a fast implementation of virtual environments.

Virtualbox [9] is cross-platform virtualization tool. Virtual machines are able to run an operating system within another operating system. The application imitates computer architecture and the user experience is the same as a dedicated hardware.

This software can deploy both x86 and AMD64/Intel 64bits architectures and the user can configure the computing capacity and memory, storage and network resources and performance according on the needs of the project.

A shared folder between the host and guest machines was defined. This way the development could be done comfortably on the host machine, while tests (and all installed dependencies) were on the guest machine.

2.4.1.2. Git version control

Even in small applications it is necessary to use some form of version control. Git [10] works in linux and windows, and allows to easily manage separated

branches when needed, and synchronize different repositories without needing a central server.

2.4.1.3. Editors

No specific development environment is needed for html and javascript, just a text editor, for no compiling is performed.

In the virtual linux server VIM [11] was used to do minor editings.

Most of the development was performed on the Windows host, with the new Visual Studio Code. It is a simple editor, with syntax highlighting and good handling of folders. It also has integrated Git support.

2.4.2. Libraries

In this section there is a brief description of each used libraries and its function:

2.4.2.1. JQuery

jQuery [12] is an open-source JavaScript framework for web development. The solutions are built by plugins and widgets. It uses CSS selectors to access and manipulate HTML elements, and wrapping them into programming methods.

jQuery works across all major browsers wrapping in a simple API the differences that may exist. It simplifies the handling of events, and allows to easily comply with the principles of separation of content, presentation and functionality.

In this project, jQuery has been basically used to separate the handling of user events, like clicks on buttons and movement over the canvas, in what it is called unobtrusive JavaScript. This way, the html is kept clean and easier to maintain.

2.4.2.2. Node.JS libraries

Node.Js [13] is a JavaScript runtime environment for developing server-side web applications. The N-open-source cross platform is based on an event-driven, non-blocking I/O model suitable for scalable real time web applications.

In this application, Node.JS was used to develop the backend, a small server that spans less than 40 lines of code called *server.js*. The ‘chat’ application needed to handle the WebRTC signaling is developed using socket.io, a library that handles communication between the browser and the server using the WebSocket protocol.

2.4.2.3. JSON

JavaScript Object Notation (JSON) [14] is a lightweight data-interchange text format and is completely language independent. It defines structures for all common data formats and supports storing and exchanging information among applications.

Using JSON it is possible to serialize objects to send from end to end. This allows to easily send and share the user event data the application needs to replicate the user actions on both the local and remote browser.

2.4.2.4. HTML5 Canvas API

The HTML5 element <canvas> [15]can be used to programmatically draw graphics inside the browser. It is a very powerful element that allows to load and process images in real time, perform multiple image composition and transformation, even video processing.

It also allows to draw lines, combined shapes, text, etc. In this application, only the simplest of all canvas functionalities has been used, the ability to draw lines.

An empty canvas is fully transparent. This allows to place the canvas element over any other element in the web page and draw over it. In our case the video element. We have set the same size for both elements: 800x450 to fit devices with similar screen dimensions to an iPad.

The existing whiteboard snippet, which is already design in the IVF application, captured all mouse or touch events (mouse in desktop, touch in touch devices like tablets or iPad). When the user touches the canvas surface the application goes into drawing mode. While the user moves its finger a line to the new coordinates is drawn. When the user lifts the finger, the application exits draw mode.

As the canvas works with absolute coordinates, and canvases on both ends have the same dimensions pixel-wise, it is pretty straightforward to synchronize the drawing on both ends by sending the coordinates used to draw a line to the remote browser, indicating that a line is to be drawn there too.

The fact that we are not really drawing over the video but on a superimposed element makes it possible (and easy!) to draw not only over static images but also video in movement.

2.4.2.5. HTML5 Video API

The HTML5 <video> [16] element allows to present a video to the user, and to control the video playback programmatically. It is possible to move the playhead to a specific moment inside the video length, to control the playback speed, pause, even play backwards.

The existing IVF application automatically presents the last recorded video to the user, and allows the user to view it, pause it, move frame by frame in any direction, or make time defined jumps (like 5 seconds back, to review the same action once and again). It should be taken into account that the frame by frame movement is not supported by the video element and it is implemented as 40ms jumps, for a 25 fps video.

From the remotization point of view, all those actions could be reduced to two: toggle play/pause, and move the playhead to a specific instant. When the user pushes the play button, the video playing state is toggled at the local browser and the same command is sent to the remote browser. Also, any action that implies a jump to a specific instant is replied on both ends.

Network lag times can introduce an offset between the remote and local playhead. When the user pauses the video locally and this command is sent to the remote browser, both videos might be stopped at different frames. Not just might, but almost surely will. This can be solved correcting the playhead position each time a pause command is issued. Each time the user pauses the video locally, the remote video element is told to pause, and then jump to the correct frame.

CHAPTER 3. Implementation details

Once the implementation is finished and all the requirements are accomplished, we proceed to explain the different functionalities of our application. Note that the solution described in this project can be considered as a proof of concept to demonstrate its feasibility, before a final production version is developed.

3.1. Explain usage

In this section, the main functionalities of the developed software are described. As we said earlier, the developed proof of concept is a web application and has to be accessed through a browser. It can be accessed from tablets, iPads or any computer. And to verify its performance, it must be running *server.js*, as mentioned in section 2.4.2.2, used to emulate a backend.

Fig. 3.1 Proof of concept screenshot

As shown in Fig. 3.1, there is a preloaded video for testing, if the application would be working in a real system as described in Fig. 2.3, the IVF will show the last recorded video.

Note there is no need to establish a remote connection to another user for the whiteboard and the video to work. You can perform analysis on your own.

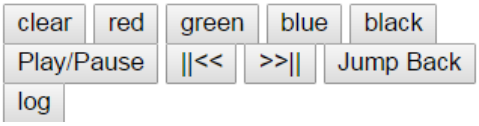

Fig. 3.2 Controls

Figure 3.2 shows the detail of the control buttons. The first row contains the controls for drawing over the video. Drawing works with static image and moving video. The user can choose between 4 colors: black, red, green and blue. Besides, there is a clear button to erase the current drawing.

The second row contains the playback controls, which allow the user to play or pause the video, move forward or rewind frame by frame and the Jump back button that rewinds the video to the start of the last move.

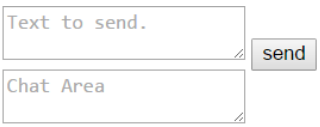

Fig. 3.3 Chat interface

Below the controls, there is a chat interface to communicate both users. You can see a screenshot in Fig. 3.2 WebRTC is able to send audio, video or just data. We choose to implement a chat interface in this project, but the customer product allows audio calling.

At the last row, there is the log button; it does not log into the video server but allows the establishment of the WebRTC session between the two browsers. The caller defines a room-word and the remote user has to type the same word to start the WebRTC communication. If another user tries to access once the channel is set, they cannot see our session because, as said in previous chapters, WebRTC uses pure peer-to-peer architecture. In figure 3.3, we follow the signaling, in our example it is called “test”. Note that all the signalig process is done through the server.

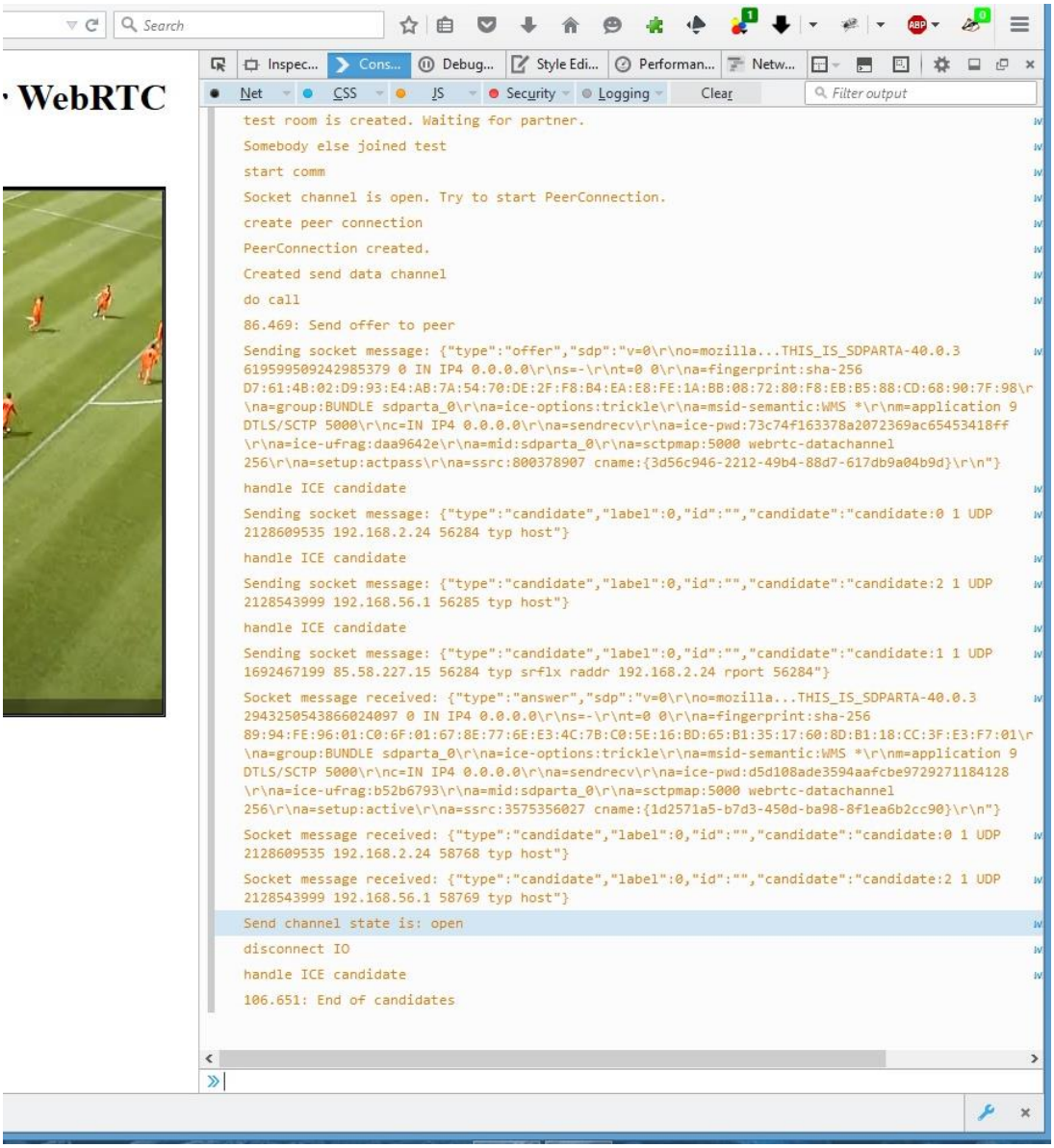

Fig. 3.4 Signaling

3.2. Challenges faced during the implementation

During the development of the application, several problems have been faced, some of them due to the nature of the application -networks are never ideal, Internet much less- and some of them due to the fact that we are using a new technology that is still being developed, and developed by different actors.

The main problems faced were browser support inconsistencies and network latencies, and the need to keep synchronized not only both ends of the communication but also the two elements that form the application, video and canvas.

There is also the key problem of extensibility, that is, which features would this application be missing?

3.2.1. Browser support. WebRTC

Several actors take part in the development of WebRTC. And from a developer's point of view, at this stage there are two main concerns, what functionalities does this new standard provide and at which stage of adoption is each one of the main browsers.

The fact that both Google and Mozilla are backing the development of the standard helps the early adoption in Chrome and Firefox, and fortunately, in contrast to the browser wars of old times, they are collaborating. But there were three main differences that kept the first versions of our application from working in Firefox. Truth be said, the modified code that worked in Firefox worked in Chrome as well.

3.2.1.1. Deprecated constraints format

When “Creating an offer” to initiate the establishment of a WebRTC communication, some constraints have to be passed as parameter. These constraints are passed as a JSON object, the format of which has changed. Chrome accepted the deprecated old format, while Firefox required the new correct format to be used.

The old format separated mandatory and optional parameters, while the new format does not.

Old:

```
var constraints = {
      'optional': [],
      'mandatory': {'MozDontOfferDataChannel': true}};
```

New:

```
var constraints = {'MozDontOfferDataChannel': true};
```

3.2.1.2. Callback functions as mandatory parameters

At the same *createOffer* function, and also when creating the answer, two parameters are used to define the callback functions for Success and Error. Though it would not be acceptable on the final product, at this stage graceful handling of errors was not a requirement -an error message or exception on the console was more than enough- therefore *null* was passed as Error callback parameter. Chrome accepted this behavior, while Firefox enforced the requirement of a callback function to be defined.

Chrome accepts (not recommended for production code):

```
peerCom.createOffer(
      setLocalAndSendMessage,
      null,
      constraints);
```

Firefox requires:

```
peerCom.createOffer(
      setLocalAndSendMessage,
      errorOnCreateOffer,
      constraints);
```

3.2.1.3. Event handling in callback functions

Some callback functions when called are passed parameters. For example, callback functions used to handle events are passed the triggering event as a parameter. By convention *event* or *e* is used to denote the event. Once again, Chrome accepted a more relaxed -some might say incorrect- coding allowing the event parameter definition to be implicit and considering an undefined event variable used on the body of the callback function as the passed event, while Firefox enforced correctness, requiring the name of the parameter to be defined on the callback function definition.

Chrome (implied *event*):

```
function handleIceCandidate() {
      trace('Handle ICE Candidate.');
      if (event.candidate) {
            ...
      }
      ...
}
```

Firefox (explicit *event*):

```
function handleIceCandidate(event) {
      trace('Handle ICE Candidate.');
      if (event.candidate) {
            ...
      }
      ...
}
```

3.2.2. Browser support. H.264

Another item of consideration, though in this case it is imposed by the existing application onto which we are building, is the codec video support in the browser. The application requires the browser to support video in h.264 over mp4 format.

The HTML5 standard does not require the video tag to support specific codecs, delegating on the browser which are supported and which not. Due to the non-free nature of h.264, support was not widespread, and mostly relied on the platform the browser was running on. For example, a year ago Firefox on Windows was able to work with h.264, while on Linux or OSX it did not.

Nowadays, h.264 is more and more supported. In any case, as already stated, this requirement is imposed.

3.2.3. Max: Resolution, fps while doing whiteboarding, what's the bottleneck

The application lifecycle has two very separate phases that have to be taken into account when looking for the usability limits.

In the first phase, the video is loaded from the server to the device, or the necessary buffering is performed. In this phase video quality, resolution and length will have an important impact on loading or buffering times. The higher the weight of the video, the higher the waiting time for the user. The accepted weight will be limited by the network capabilities; it is not the same downloading a 100MB video from a server on a local high speed LAN, than trying to do the same through a 3G mobile network.

In any case, this is outside the scope of this project. The video quality and length is defined by the underlying existing application, and ultimately is up to the user, it is not the same to perform tactical analysis of a football team where rough field positions are enough, than trying to adjust the hand position of a synchro swimmer, where a finger out of place may be the detail that leaves the athlete out of the medals.

The second phase is the analysis, including video review and whiteboarding. This is the phase we are covering, and it is a requirement for this project that, during this analysis, video and whiteboard have to be synchronized at both ends of the communication, the trainer browser and the athlete browser.

The fact that the solution adopted implies only transmitting the commands the user issues but not trying to share the screen or stream images end to end in any way, helps in avoiding network bandwidth issues. The necessary bandwidth is completely negligible in front of what is needed to load the video at the beginning, even a small video. Our tests showed almost no impact on the computer network behavior, even if the user is trying to paint furiously all over the video (not to probable scenario). Network latency and communication delays have to be taken on account for they may impact usability (see section 3.2.3).

The worst case would be the scenario whereby firewall and network configuration forces the application to use TURN servers. In this case, P2P is not working and all messages go through the server. Even in this case server load would be negligible. This is not an application that needs to support massive number of users.

3.2.4. Network impact on usability

Apart from the bandwidth analyzed in section 3.2.1, the network might introduce lag times and communication delays that do have impact on the application usability. Those can be classified into two categories, issues on local perception and usability -due to the remote layer added, the local user perceives problems caused by those delays- and desynchronization problems between both ends of the application.

3.2.4.1. Local issues

JavaScript handles all the application execution in a single thread. Sending events and commands to the remote browser introduces processing times and waiting times in the local execution flow.

For example, all whiteboarding actions are responses to a user event (touch or mouse) over the canvas. When the user moves a finger over the canvas, he expects a line to be drawn following it. At the same time we want this line to be drawn over the remote canvas. There are two approaches to this, local first or remote first.

Remote first:

1. Send command to draw remote line
2. Draw local line

Local first:

1. Draw local line
2. Send command to draw remote line

If the command to draw the remote line is sent *before* drawing the local line, the user notices the delay between the movement of the finger and the line appearing, having the perception that the application is slow and not responding appropriately. On the other hand, if the line is drawn prior to sending the command, the remote user has no delay perception, and the local user feels the application responds smoothly to the finger movement.

Local first approach was used for all commands.

```
function(obj, event, isRemote) {
...
// Local line
this.context.lineTo(pos.x,pos.y);
this.context.stroke();
// If connected, remote line
if(this.remote.isReadyCom && !isRemote) {
            this.remote.sendData('{"type" : "whiteboard",
                                  "function" : "drawingLine",
```

```
                                    "params" : ' + JSON.stringify(pos) +
                              '}');
      }
      }
```

3.2.4.2. Synchronization problems

Being a remote network application, some end to end delay is to be expected, and within certain levels, the user won't complain even if these delays were perceived -which will seldom occur, the remote user is remote, and won't see the finger moving-.

But this delay, in addition to be non-negligible, is unpredictable. It is not always the same, nor can be known in advance. This means that consecutive orders will arrive with different traveling times, introducing probable synchronization problems.

These synchronization problems become evident when an action is relative to the time the user triggers it. When the user pauses the video now, this command arrives to the remote browser a little bit later. This would not be a problem if the delay was always the same, as play and pause would arrive with the same offset, the video would pause a little bit later but in the exact same static image. The problem is that, as the delay is variable and unpredictable, the offset will be different, and the video will be paused in a different frame.

This can be a minor problem for an overview of tactical analysis if kept within a certain range, no one will notice a two or three frames difference, but it is total disaster in technical analysis especially in certain sports. For instance, to determine line-calling in tennis.

This problem can be solved using absolute references for the remote commands. When sending pause, the command will always be followed by an immediate command indicating in which precise video time the image has to be frozen. This may generate a small jump on the receiving end that will be perceivable but totally acceptable.

```
      if(this.remote.isReadyCom && !isRemote) {
            console.log("Send to remote");
            this.remote.sendData('{
                              "type" : "videoplayer",
                              "function" : "togglePlay",
                              "params" : ""}');
            // There will be a delay. We can adjust with currentTime
            if (this.videoplayer.paused) {
                  var current = this.videoplayer.currentTime;
                  this.remote.sendData('{
                              "type" : "videoplayer",
                              "function" : "moveTo",
                              "params" : ' + current + '}');
            }
      }
```

3.2.5. Evaluation in multiple environments

It is a requirement that the application runs in different devices and operative systems. Though being a web application that fully lives inside the browser helps, it has to be tested for differences may appear between browsers, and even among the same browser between different OS.

WebRTC, as of now, imposes a restriction to work over Chrome or Firefox. As this restriction was already present for some other features in the original application, this restriction is totally accepted.

Though several devices have been tested (Windows PC, Mac computer, Android handheld device, etc.). Testing of any combination is recommended before going production with it.

3.3. Possible short-term extensions

During the internal tests of the application, we have detected some possible extensions that could be developed in the short term.

In this application only the data channel of WebRTC has been used. WebRTC is designed to work with audio and video streams that could be easily added to the application. This would be subject to a prior analysis on user impact. Not only at a feature interest and usability, but considering that a real-time audio stream and specially a video stream would require several orders of magnitude more bandwidth than the application needs now.

Also, WebRTC allows full bidirectional communication, and it has been implemented in the application. Both users can play the video or draw over it. This can be a desired feature, or something to be restricted.

3.3.1. Audio stream

Adding a real-time audio stream to the application would be a much appreciated feature, allowing the coach to explain to the athletes the action, with much more comfort and easy feedback than the current chat solution.

3.3.2. Video stream

Though it might sound appealing at first, it is not clear what advantages might bring. The real focus of interest for the athlete is the video that is being reviewed. Adding the face of the coach side by side would only add distraction and reduce space for the video. The coach already has access to the cameras on the training center, so the video of the athlete would add nothing.

On the other hand in some cases it could be useful for the coach to be able to show to the player the correct technique demonstraiting by himself.

It should be taken on account that some factors out of the real use must be considered. In sports it is quite often that the user of the system and the key decision maker for the purchase (the *money* man) are different persons. And definitely a video chat is something that, as already stated, sounds appealing.

3.3.3. Bidirectionality

It might be a good idea to add limits to the bidirectionality, leaving to the coach the decision whether the athlete can collaborate or not (or whether is not an athlete but a second coach).

CHAPTER 4. Conclusion and next steps

4.1. Conclusion

This thesis presents our solution for Remote Interactive Video Feedback application, although we only show the proof of concept. It has been a long process, as outlined in the introduction, which has included requirement gathering, design, implementation, and testing with customers on the completed product.

As mentioned in the introduction, the application had to work with as little prior installation as possible, to be flexible in where it was used, as well as support multiple platforms and interaction modes, from a PC to a tablet. The application needed to provide video playback synchronization, and still support drawing tactical analysis graphs on it in, so that both peers could interact. The application needed to “feel agile and light” and expressive enough for the needs of the task.

Our proposal was to develop a Web Application that injects an HTML 5 Canvas that’s synchronized using the WebRTC protocol. The develop environment was build combining Vagrant and VirtualBox, Git has been used for version control and VIM to edit code. Several JavaScript libraries have been used: jQuery to separate the handling of user events; NodeJS to emulate a backend with few lines of code and socket.io library and WebSocket for signaling; JSON to ease send and share the user event data that has to been replicated in both browser. And final, but not least, the two HTML5 elements: canvas and video superimposed.

The choice of technologies has proven to be very adequate for the particular scenario we had in mind, and the fact so much of it was based on open standards means it was relatively easy to find documentation and support for it during the work. It also means that the result is time tested platform to grow and build upon. The web has proven once again its flexibility, and this experience has been eye opening in terms of how much potential it still has, and how many more applications fit the capabilities it provides.

The results have been very satisfying, not only in terms of the customer’s very positive feedback, but also as a growth opportunity and learning experience.

We look forward to the widespread usage of the application, as well as its evolution, including the next steps outlined below.

4.2. Next steps

As often happens when customers are satisfied, we already have numerous requests for improvements on the product, on top of the changes we have already been meaning to perform. These are the most important ones:

- Polishing the UI design and improving the User Experience
- Add to the proof of concept audio streaming from the coach
- Support to record the streams of the whiteboard interaction
- Advanced, more expressive drawing artifacts for specific sports
- Playback of multiple simultaneous videos, to illustrate the effect of different tactics on different occasions
- Add responsiveness: modify R-IVF to be able to resize the video/canvas element to adjust to device screen size.

4.3. Environment Impact

Elite training centers already have IP cameras and servers for tactic review and training. In this context, adding support for remote participation is a relatively low cost.

Additionally, the added energy budget is negligible, except for the fact that multiple peers means multiple devices and the cost of transferring the information remotely. We believe, however, that this is dwarfed by the huge savings in travel that remote collaboration brings, not just in terms of manpower and psychological toll, but also in the many hours of flight, train and car transportation that will be saved, and the perhaps small, yet positive contribution this will have to the environment.

Bibliography

[1] S. Dutton, "Getting Started with WebRTC - HTML5 Rocks," February 2014. [Online]. Available: http://www.html5rocks.com/en/tutorials/webrtc/basics/.

[2] S. Dutton, «WebRTC in the real world: STUN, TURN and signaling,» November 2013. [En línea]. Available: http://www.html5rocks.com/en/tutorials/webrtc/infrastructure/.

[3] Google Developers, «Real-time communication with WebRTC: Google I/O 2013,» May 2013. [En línea]. Available: https://www.youtube.com/watch?v=p2HzZkd2A40.

[4] Refsnes Data, «HTML Canvas Tutorial,» [En línea]. Available: http://www.w3schools.com/canvas/default.asp. [Último acceso: July 2015].

[5] W3C, «Scalable Vector Graphics (SVG) 1.1 (Second Edition),» [En línea]. Available: http://www.w3.org/TR/SVG/. [Último acceso: June 2015].

[6] Object Management Group, Inc, «Unified Modeling Language (UML),» [En línea]. Available: http://www.uml.org/. [Último acceso: August 2015].

[7] HashiCorp, «Vagrant Blog,» [En línea]. Available: https://www.vagrantup.com/. [Último acceso: May 2015].

[8] HashiCorp, "Development environments made easy.," [Online]. Available: https://www.vagrantup.com/. [Accessed May 2015].

[9] Oracle, "VirtualBox – Oracle VM VirtualBox," [Online]. Available: https://www.virtualbox.org/wiki/VirtualBox. [Accessed May 2015].

[10] Software Freedom Conservancy, "Git," [Online]. Available: http://www.git-scm.com/. [Accessed May 2015].

[11] "Welcome home: vim online," [Online]. Available: http://www.vim.org/. [Accessed June 2015].

[12] The jQuery Foundation, «How jQuery Works,» [En línea]. Available: http://learn.jquery.com/about-jquery/how-jquery-works/. [Último acceso: June 2015].

[13] Node.js Foundation, «Node.js,» [En línea]. Available: https://nodejs.org/en/. [Último acceso: June 2025].

[14] Refsnes Data, « JSON Tutorial,» [En línea]. Available: http://www.w3schools.com/json/. [Último acceso: July 2015].

[15] Html5CanvasTutorials, «HTML5 Canvas Tutorials,» [En línea]. Available: http://www.html5canvastutorials.com/. [Último acceso: July 2015].

[16] Refsnes Data, «TML5 Video,» [En línea]. Available: http://www.w3schools.com/html/html5_video.asp. [Último acceso: August 2015].

Annex

Here we will find all the different coding modules we have developed till the full functionality for the proof of concept was achieved.

Gruntfile.js

Grunt configuration file. It defines the tasks to be performed by Grunt to, in this case, lint the HTML and the Javascript files when they are modified. It is not properly part of the application.

```
/*global module:false*/
module.exports = function(grunt) {

   // Project configuration
   grunt.initConfig({
       htmllint: {
           files: 'src/*.html'
       },
       watch: {
           jsfiles: {
               files: ['src/js/**/*.js'],
               tasks: 'jshint:development'
           },
           htmlfiles: {
               files: ['src/*.html'],
               tasks: 'htmllint'
           }
       },
       jshint: {
           // Default
           options: {
               curly: true,
               eqeqeq: true,
               immed: true,
               latedef: false,
               newcap: true,
               noarg: true,
               sub: true,
               undef: true,
               unused: true,
               boss: true,
               eqnull: true,
               browser: true,
               devel: true,
               globals: {
                   // don't complain about jQUery and Undersocre
                   $: false,
                   jQuery: false,
                   _V_: false,
                   _: false
               }
           },
           development: ['grunt.js', 'src/js/**/*js']
       }
   });
```

```
   grunt.loadNpmTasks('grunt-html');
   grunt.loadNpmTasks('grunt-contrib-jshint');
   grunt.loadNpmTasks('grunt-contrib-watch');

   grunt.registerTask('default', 'watch');
   grunt.registerTask('lint', ['htmllint', 'jshint:development']);
};
```

Package.json

Defines the application and its dependencies. node-static and socket.io are required to implement the server for the WebRTC Siganling.

```
{
 "name": "synchvideoboard",
 "version": "1.0.0"  "description": "Remote Interactive Whiteboard Web
Application with Synchronized Video Playback",
 "main": "server.js",
 "scripts": {
      "start": "node server.js"
 },
 "engines": {
      "node": "0.12.7"
 },
 "keywords": [
      "video", "WebRTC", "canvas"
 ],
 "author": "",
 "license": "MIT",
 "dependencies": {
      "node-static": "~0.7.1",
      "socket.io": "~0.9.16"
 }
}
```

Server.js

Main server side file of the application. It has to be started with node, to serve the static html files and handle the websocket messages used for WebRTC signaling.
This is the complete version.

```
var nodestatic = require('node-static');
var http = require('http');
var file = new(nodestatic.Server)();
var app = http.createServer(function (req, res) {
   file.serve(req, res);
}).listen(2013);

var io = require('socket.io').listen(app);
io.sockets.on('connection', function (socket) {

   socket.on('message', function (message) {
```

```
        socket.broadcast.to(socket.room).emit('message', message);
    });

    socket.on('join', function (room) {
        var numClients = io.sockets.clients(room).length;
        if (numClients == 0) {
            // New rooom
            socket.join(room);
            socket.room = room;
            socket.emit('created', room);
        } else if (numClients == 1) {
            // Existent
            io.sockets.in(room).emit('join', room);
            socket.join(room);
            socket.room = room;
            socket.emit('joined', room);
        } else {
            // Full
            socket.emit('full', room);
        }
    });

    socket.on('disconnect', function() {
        //io.sockets.in(socket.room).emit('left', socket.room);
        //socket.room = null;
    });
});
```

Server_flat.js

Main server side file of the application. It has to be started with node. This is the first version. Only serves static files.

```
var nodestatic = require('node-static');
var http = require('http');
var file = new(nodestatic.Server)();
var app = http.createServer(function (req, res) {
   file.serve(req, res);
}).listen(2013);
```

Server_io.js

Main server side file of the application. It has to be started with node. This version serves static files and handles a simple chat through socket.io.

```
var nodestatic = require('node-static');
var http = require('http');
var file = new(nodestatic.Server)();
var app = http.createServer(function (req, res) {
   file.serve(req, res);
}).listen(2013);

var io = require('socket.io').listen(app);
io.sockets.on('connection', function (socket) {
```

```
   socket.on('message', function (message) {
       // broadcast not a good idea
       //socket.broadcast.emit('message', message); // A tooodos
       // socket.emit('message', message); // Solo vuelve al mismo emisor
       socket.broadcast.to(socket.room).emit('message', message); // A
socket.room, si existe
   });

   socket.on('join', function (room) {
       var numClients = io.sockets.clients(room).length;
       if (numClients == 0) {
           // New rooom
           socket.join(room);
           socket.room = room;
           socket.emit('created', room);
       } else if (numClients == 1) {
           // Existent
           io.sockets.in(room).emit('join', room);
           socket.join(room);
           socket.room = room;
           socket.emit('joined', room);
       } else {
           // Full
           socket.emit('full', room);
       }
   });

   socket.on('leave', function(room) {
       socket.leave(socket.room);
       io.sockets.in(socket.room).emit('left', room);
       socket.room = null;
   });

   socket.on('disconnect', function() {
       io.sockets.in(socket.room).emit('left', socket.room);
       socket.room = null;
   });
});
```

Index_canvas.html

Main page of the application. Holds the HTML code, and calls all required javascript files.
This version only calls the canvas functionality.

```
<!doctype html>
<html lang="en">
   <head>
       <meta charset="utf-8">
       <title>Canvas Test</title>
       <style>
           canvas {
               border: 3px solid #000000;
               /*
               Width and height of canvas element are not properly defined
here
```

```
                width: 800px;
                height: 400px;
                */
                }
        </style>
    </head>
    <body>
        <body>
        <h1>Simple Canvas functionality</h1>
        <p> Needs no special server capabilities</p>
        <canvas id="whiteboard">
            Please, check your browser &lt;canvas&gt; support.
        </canvas>
        <br style="clear: both">
        <input type="button" value="clear" id="clear" />
        <input type="button" value="red" id="red" />
        <input type="button" value="green" id="green" />
        <input type="button" value="blue" id="blue" />
        <input type="button" value="black" id="black" />

        <script src="lib/jquery/jquery.min.js"></script>
        <script src='js/whiteboard.js'></script>

        <script>
            var whiteboard = com.knovz.whiteboard.init({"canvasId" :
"whiteboard", "linkEvents" : true});
            // canvas size can be set here, or hardcoded in the element, but
NOT in CSS
            whiteboard.setCanvasSize(800,450);
            $("#clear").click(function(event) {whiteboard.clear()});
            $("#red").click(function(event) {whiteboard.setColor("#ff0000")});
            $("#green").click(function(event)
{whiteboard.setColor("#00ff00")});
            $("#blue").click(function(event)
{whiteboard.setColor("#0000ff")});
            $("#black").click(function(event)
{whiteboard.setColor("#000000")});
        </script>
    </body>
</html>
```

Index_io.html

Main page of the application. Holds the HTML code, and calls all required javascript files.
This version is used to test socket.io functionality.

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Test Socket IO</title>
        <script src='socket.io/socket.io.js'></script>
    </head>
    <body>
        <h1>Test simple IO messaging functionality</h1>
```

```
       <p> You must be running server_io.js on the server for this to
work</p>
       <textarea id="dataChannelSend" disabled placeholder="Text to
send."></textarea>
       <div id="buttons">
           <button id="startButton">Log</button>
           <button id="sendButton">Send</button>
           <button id="closeButton">Stop</button>
           <button id="disconnectButton">Fully Disconnect</button>
       </div>

       <script>
function trace(text) {
   console.log((performance.now() / 1000).toFixed(3) + ": " + text);
}

var socket = io.connect();

var startButton = document.getElementById("startButton");
var sendButton = document.getElementById("sendButton");
var closeButton = document.getElementById("closeButton");
var disconnectButton = document.getElementById("disconnectButton");
startButton.disabled = false;
sendButton.disabled = true;
closeButton.disabled = true;
startButton.onclick = logToRoom;
sendButton.onclick = sendData;
closeButton.onclick = closeDataChannels;
disconnectButton.onclick = disconnectIO;

var isInitiator = false;
function logToRoom() {
   room = prompt("Enter room name");
   if (room !== "") {
       socket.emit('join', room);
   }
}

function closeDataChannels() {
   socket.emit('leave', room);
   isInitiator = false; // you are out now
   startButton.disabled = false;
   sendButton.disabled = true;
   closeButton.disabled = true;
}

function disconnectIO() {
   socket.disconnect();
}

socket.on('full', function(room) {
   trace('room is full');
});

socket.on('created', function(room) {
   isInitiator = true;
   startButton.disabled = true;
   closeButton.disabled = false;
```

```
   trace('room is created');
});

socket.on('joined', function(room) {
   isInitiator = false;
   startButton.disabled = true;
   closeButton.disabled = false;
   trace('you joined room');
   startComm();
});

socket.on('left', function(room) {
   isInitiator = true;  // you are alone now
   trace('your partner left the room');
});

socket.on('join', function(room) {
   trace('somebody else joined');
   startComm();
});

socket.on('message', function(message) {
   trace('Received: ' + message);
});

function startComm() {
   sendButton.disabled = false;
   document.getElementById("dataChannelSend").disabled = false;
}

function sendData() {
   var data = document.getElementById("dataChannelSend").value;
   socket.emit('message', data);
   trace('Sent: ' + data);
}
       </script>
   </body>
</html>
```

Index_canvas_remote.html

Main page of the application. Holds the HTML code, and calls all required javascript files.
This version calls the canvas with remote functionality.

```
<!doctype html>
<html lang="en">
   <head>
       <meta charset="utf-8">
       <title>Canvas Remote Test</title>

       <script src='socket.io/socket.io.js'></script>
       <script src='lib/adapter.js'></script>

       <style>
```

```
            canvas {
                border: 3px solid #000000;
                /*
                Width and height of canvas element are not read if defined
here
                width: 800px;
                height: 450px;
                */
            }
            #chatArea {
                height : 200px;
            }

        </style>

    </head>
    <body>
        <body>
        <h1>Canvas with remote functionality over WebRTC</h1>
        <p> You must be running server.js on the server for this to work</p>
        <canvas id="whiteboard">
            Please, check your browser &lt;canvas&gt; support.
        </canvas>
        <br style="clear: both">
        <input type="button" value="clear" id="clear" />
        <input type="button" value="red" id="red" />
        <input type="button" value="green" id="green" />
        <input type="button" value="blue" id="blue" />
        <input type="button" value="black" id="black" />
        <br style="clear: both">
        <input type="button" value="log" id="log" />
        <br style="clear: both">
        <textarea id="chatSend" placeholder="Text to send."></textarea>
        <input type="button" value="send" id="send" />
        <br style="clear: both">
        <textarea id="chatArea" placeholder="Chat Area"></textarea>

        <script src="lib/jquery/jquery.min.js"></script>
        <script src='js/whiteboard.js'></script>
        <script src='js/whiteboard_remote.js'></script>

        <script>
            var whiteboard = com.knovz.whiteboard.init({"canvasId" :
"whiteboard", "linkEvents" : true});
            // canvas size can be set here, or hardcoded in the element, but
NOT in CSS
            whiteboard.setCanvasSize(800,400);
            $("#clear").click(function(event) {whiteboard.clear()});
            $("#red").click(function(event) {whiteboard.setColor("#ff0000")});
            $("#green").click(function(event)
{whiteboard.setColor("#00ff00")});
            $("#blue").click(function(event)
{whiteboard.setColor("#0000ff")});
            $("#black").click(function(event)
{whiteboard.setColor("#000000")});

            $("#log").click(function(event) {whiteboard.remote.logToRoom();});
            $("#send").click(function(event) {
```

```
                whiteboard.remote.sendChatMessage(chatSend.value);
            });
            whiteboard.remote.chat = chatArea;
        </script>
    </body>
</html>
```

Index_canvas_video.html

Main page of the application. Holds the HTML code, and calls all required javascript files.
This version adds video functionality to the canvas. Only local.

```
<!doctype html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Canvas and Video Test</title>
        <style>
            #videoarea {
                border: 3px solid #000000;
                position: relative;
            }
            #videoplayer {
                border: 1px solid #666699;
            }
            #whiteboard {
                border: 1px solid #666666;
                position: absolute;
                top: 0px;
                left: 0px;
                /*
                Width and height of canvas element are not read if defined
here
                width: 800px;
                height: 400px;
                */
                }
        </style>

    </head>
    <body>
        <body>
        <h1>Simple Canvas and Video functionality</h1>
        <p> Needs no special server capabilities</p>
        <div id="videoarea">
            <video id="videoplayerel" preload="metadata">
            </video>
            <canvas id="whiteboard">
                Please, check your browser &lt;canvas&gt; support.
            </canvas>
        </div>
        <br style="clear: both">
        <input type="button" value="clear" id="clear" />
        <input type="button" value="red" id="red" />
        <input type="button" value="green" id="green" />
        <input type="button" value="blue" id="blue" />
```

```
        <input type="button" value="black" id="black" />

        <br style="clear: both">
        <input type="button" value="Play/Pause" id="togglePlay" />
        <input type="button" value="||<<" id="stepback" />
        <input type="button" value=">>||" id="stepfwd" />
        <input type="button" value="Jump Back" id="jumpback" />

        <script src="lib/jquery/jquery.min.js"></script>
        <script src='js/whiteboard.js'></script>
        <script src='js/videoplayer.js'></script>

        <script>
            // Whiteboard
            var whiteboard = com.knovz.whiteboard.init({"canvasId" :
"whiteboard", "linkEvents" : true});
            // canvas size can be set here, or hardcoded in the element, but
NOT in CSS
            whiteboard.setCanvasSize(800,450);

            // Videoplayer
            var videoplayer = com.knovz.videoplayer.init({"playerId" :
"videoplayerel"});
            videoplayer.setPlayerSize(800,450);
            videoarea.style.width = '800px';
            videoarea.style.height = '450px';

            // External controls Canvas
            $("#clear").click(function(event) {whiteboard.clear()});
            $("#red").click(function(event) {whiteboard.setColor("#ff0000")});
            $("#green").click(function(event)
{whiteboard.setColor("#00ff00")});
            $("#blue").click(function(event)
{whiteboard.setColor("#0000ff")});
            $("#black").click(function(event)
{whiteboard.setColor("#000000")});

            // External controls Video
            $("#togglePlay").click(function(event)
{videoplayer.togglePlay()});
            $("#stepback").click(function(event)
{videoplayer.moveStepBack()});
            $("#stepfwd").click(function(event) {videoplayer.moveStepFwd()});
            $("#jumpback").click(function(event) {videoplayer.move(-5)});

            // load video
            videoplayer.load({"src" : "video/sample.mp4", "poster" :
"video/poster.jpg"});
        </script>
    </body>
</html>
```

Index_canvas_video_remote.html

Main page of the application. Holds the HTML code, and calls all required javascript files.

This is the complete version, with video and whiteboarding, with remote functionality.

```
<!doctype html>
<html lang="en">
   <head>
       <meta charset="utf-8">
       <title>Test Socket IO</title>
       <script src='socket.io/socket.io.js'></script>
   </head>
   <body>
       <h1>Test simple IO messaging functionality</h1>
       <p> You must be running server_io.js on the server for this to
work</p>
       <textarea id="dataChannelSend" disabled placeholder="Text to
send."></textarea>
       <div id="buttons">
           <button id="startButton">Log</button>
           <button id="sendButton">Send</button>
           <button id="closeButton">Stop</button>
           <button id="disconnectButton">Fully Disconnect</button>
       </div>

       <script>
function trace(text) {
   console.log((performance.now() / 1000).toFixed(3) + ": " + text);
}

var socket = io.connect();

var startButton = document.getElementById("startButton");
var sendButton = document.getElementById("sendButton");
var closeButton = document.getElementById("closeButton");
var disconnectButton = document.getElementById("disconnectButton");
startButton.disabled = false;
sendButton.disabled = true;
closeButton.disabled = true;
startButton.onclick = logToRoom;
sendButton.onclick = sendData;
closeButton.onclick = closeDataChannels;
disconnectButton.onclick = disconnectIO;

var isInitiator = false;
function logToRoom() {
   room = prompt("Enter room name");
   if (room !== "") {
       socket.emit('join', room);
   }
}

function closeDataChannels() {
   socket.emit('leave', room);
   isInitiator = false; // you are out now
   startButton.disabled = false;
   sendButton.disabled = true;
   closeButton.disabled = true;
}
```

```
function disconnectIO() {
   socket.disconnect();
}

socket.on('full', function(room) {
   trace('room is full');
});

socket.on('created', function(room) {
   isInitiator = true;
   startButton.disabled = true;
   closeButton.disabled = false;
   trace('room is created');
});

socket.on('joined', function(room) {
   isInitiator = false;
   startButton.disabled = true;
   closeButton.disabled = false;
   trace('you joined room');
   startComm();
});

socket.on('left', function(room) {
   isInitiator = true;  // you are alone now
   trace('your partner left the room');
});

socket.on('join', function(room) {
   trace('somebody else joined');
   startComm();
});

socket.on('message', function(message) {
   trace('Recived: ' + message);
});

function startComm() {
   sendButton.disabled = false;
   document.getElementById("dataChannelSend").disabled = false;
}

function sendData() {
   var data = document.getElementById("dataChannelSend").value;
   socket.emit('message', data);
   trace('Sent: ' + data);
}
       </script>
   </body>
</html>
```

Whiteboard.js

Javascript file holding the whiteboard functionality. Only local.

```
/*!
```

```
* com.knovz.whiteboard v1.0.0
* Licensed under MIT
*/

// Check requisites
if (typeof jQuery === "undefined") { throw new Error("com.knovz.whiteboard
requires jQuery") }

// Namespacing
var com = com || {};
com.knovz = com.knovz || {};

// whiteboard
com.knovz.whiteboard = {
	options : {},
	defaultOptions : {
		linkEvents : false,
		texts : {
			errorMsg : "ERROR"
		}
	},
	canvas : {},
	context : {},
	color : "#000000",
	lineWidth : 2,
	isDrawing : false,
	init : function(options) {
		// Options must be an object containing a canvas id
		if (!((typeof options === "object") && (typeof options.canvasId
=== "string") && (options.canvasId.length > 0))) { throw new
Error("com.knovz.whiteboard.init requires options in json format, with at
least a non empty string canvasId defining the canvas to use.") }
		// Combine parameter options with defaultOptions, into options
		this.options = $.extend(true,{},this.defaultOptions,options);
		// Get canvas and 2d context
		this.canvas = document.getElementById(this.options.canvasId);
		this.context = this.canvas.getContext("2d");
		if(this.options.linkEvents) {
			this.events.link(this.options.canvasId);
		}
		return this;
	},
	setCanvasSize : function(w, h) {
		this.canvas.width = w;
		this.canvas.height = h;
	},
	clear : function() {
		// Careful! canvas.height and canvas.width are not properly read
if CSS set.
		this.context.clearRect(0,0,this.canvas.width,this.canvas.height)
	},
	setColor : function(color) {
		this.color = color;
	},
	drawingStart : function(obj, event) {
		var pos = this.events.getPoint(obj, event);
		this.context.strokeStyle = this.color;
		this.context.lineWidth = this.lineWidth;
		this.context.beginPath();
```

```
		this.context.moveTo(pos.x, pos.y);
		this.isDrawing = true;
	},
	drawingLine : function(obj, event) {
		if(this.isDrawing) {
			var pos = this.events.getPoint(obj, event);
			this.context.lineTo(pos.x,pos.y);
			this.context.stroke();
		}
	},
	drawingEnd : function(obj, event) {
		if(this.isDrawing) {
			this.drawingLine(obj, event);
			this.isDrawing = false;
		}
	},
	events : {
		link : function(canvasid) {
			var we = com.knovz.whiteboard.events;
			if ('ontouchstart' in window) {
				/* browser with Touch Events running on touch-
capable device */
				$('#'+canvasid).bind("touchstart",function(e)
{we.canvasEvent(this,e);});
				$('#'+canvasid).bind("touchmove",function(e)
{e.preventDefault();we.canvasEvent(this,e);});
				$('#'+canvasid).bind("touchend",function(e)
{we.canvasEvent(this,e);});
			} else {
				$('#'+canvasid).bind("mousedown",function(e)
{we.canvasEvent(this,e);});
				$('#'+canvasid).bind("mousemove",function(e)
{we.canvasEvent(this,e);});
				$('#'+canvasid).bind("mouseup",function(e)
{we.canvasEvent(this,e);});
			}
		},
		touchstart : function(obj, event) {
			com.knovz.whiteboard.drawingStart(obj, event);
		},
		touchmove : function(obj, event) {
			com.knovz.whiteboard.drawingLine(obj, event);
		},
		touchend : function(obj, event) {
			com.knovz.whiteboard.drawingEnd(obj, event);
		},
		mousedown : function(obj, event) {
			com.knovz.whiteboard.drawingStart(obj, event);
		},
		mousemove : function(obj, event) {
			com.knovz.whiteboard.drawingLine(obj, event);
		},
		mouseup : function(obj, event) {
			com.knovz.whiteboard.drawingEnd(obj, event);
		},
		canvasEvent : function(obj, event) {
			com.knovz.whiteboard.events[event.type](obj, event);
		},
		getPoint : function(obj, event) {
```

```
                var target;
                if(event.originalEvent.touches) {
                        target = event.originalEvent.touches[0];
                } else {
                        target = event;
                }
                var x = Math.round(target.pageX - $(obj).offset().left);
                var y = Math.round(target.pageY - $(obj).offset().top);
                return { 'x' : x , 'y' : y };
        }
    }
};
```

Whiteboard_remote.js

Javascript file holding the remote whiteboarding functionality. Builds on whiteboard.js, and overwrites some of it's functionality.

```
/*!
* com.knovz.whiteboard.remote v1.0.0
* This is an add-in for com.knovz.whiteboard.
* Licensed under MIT
*/

// Namespacing
var com = com || {};
com.knovz = com.knovz || {};

// Check Whiteboard
if (typeof com.knovz.whiteboard === "undefined") { throw new
Error("com.knovz.whiteboard.remote requires com.knovz.whiteboard") }

com.knovz.whiteboard.remote = {
    socket : {},
    sendChannel : {},
    isInitiator : false,
    isChannnelReady : false, // Socket.io channel
    isStarted : false, // PeerCom
    isReadyCom : false,
    peerCom : {},
    chat : {},
    pc_constraints : {
        'optional': [
            {'DtlsSrtpKeyAgreement': true},
            {'RtpDataChannels': true}    // Això es per comm.
        ]
    },
    socketInit : function() {
        this.socket = io.connect();
        var socket = this.socket;
        var remote = this;  // 'this' easily means something different

        socket.on('full', function(room) {
            console.log('room is full');  // TODO show error message
        });
        socket.on('created', function(room) {
            remote.isInitiator = true;
```

```
                    console.log(room + ' room is created. Waiting for
partner.');
             });
             socket.on('joined', function(room) {
                    remote.isInitiator = false;
                    console.log('you joined room ' + room);
                    remote.isChannnelReady = true;
                    remote.startComm();
             });
             socket.on('join', function(room) {
                    console.log('Somebody else joined ' + room);
                    remote.isChannnelReady = true;
                    remote.startComm();
             });
             socket.on('message', function(message) {
                    console.log('Socket message received: ' +
JSON.stringify(message));
                    // Signaling
                    // Offer is sent from initiator
                    // and ICE candidates too
                    if (message.type === 'offer') {
                           // It should be isStarted, but...
                           if(!remote.isInitiator && !remote.isStarted) {
                                  remote.startComm();
                           }
                           remote.peerCom.setRemoteDescription(new
RTCSessionDescription(message));
                           remote.doAnswer();
                    } else if (message.type === 'answer' && remote.isStarted)
{
                           remote.peerCom.setRemoteDescription(new
RTCSessionDescription(message));
                    } else if (message.type === 'candidate' &&
remote.isStarted) {
                           var candidate = new
RTCIceCandidate({sdpMLineIndex:message.label, candidate:message.candidate});
                           remote.peerCom.addIceCandidate(candidate);
                    } else if (message === 'bye' && remote.isStarted) {
                           // TODO
                    }
             });
      },
      sendSocketMessage : function(message) {
             console.log('Sending socket message: ' +
JSON.stringify(message));
      this.socket.emit('message', message);
      },
      disconnectIO : function() {
             console.log("disconnect IO");
             this.socket.disconnect();
      },
      logToRoom : function() {
             // TODO check if it is already logged, and then un-log instead
             var room = prompt("Please enter room name");
             if (room !== "") {
                    this.socket.emit('join', room);
             }
      },
      startComm : function() {
```

```
		console.log("start comm");
		if(!this.isStarted  && this.isChannnelReady) {
			console.log('Socket channel is open. Try to start
PeerConnection.');
			this.createPeerConnection();
			this.isStarted = true;
			if(this.isInitiator) {
				this.doCall();
			}
		}
	},
	createPeerConnection : function() {
		console.log("create peer connection");
		try {
			this.peerCom = new RTCPeerConnection(null,
this.pc_constraints);
			this.peerCom.onicecandidate = this.handleIceCandidate;
			console.log('PeerConnection created.');
		} catch(e) {
			console.log('Failed to create PeerConnection: ' +
e.message);
		}

		if(this.isInitiator) {
			try {
				// Reliable Data Channels not yet supported in
Chrome
				this.sendChannel =
this.peerCom.createDataChannel("sendDataChannel", {reliable: false});
				this.sendChannel.onmessage = this.handleMessage;
				console.log('Created send data channel');
			} catch (e) {
				console.log('createDataChannel() failed with
exception: ' + e.message);
			}
			this.sendChannel.onopen =
this.handleSendChannelStateChange;
			this.sendChannel.onclose =
this.handleSendChannelStateChange;
		} else {
			this.peerCom.ondatachannel = this.gotReceiveChannel;
			console.log('Asigned receive');
		}
	},
	handleIceCandidate :  function(event) {
		console.log("handle ICE candidate");
		if (event.candidate) {
			com.knovz.whiteboard.remote.sendSocketMessage({
				type: 'candidate',
				label: event.candidate.sdpMLineIndex,
				id: event.candidate.sdpMid,
				candidate: event.candidate.candidate
			});
		} else {
			trace('End of candidates');
		}
	},
	handleSendChannelStateChange : function() {
```

```
            var readyState =
com.knovz.whiteboard.remote.sendChannel.readyState;
            console.log('Send channel state is: ' + readyState);
            // handle button states, etc...
            com.knovz.whiteboard.remote.enableMessageInterface(readyState ==
"open");
      },
      gotReceiveChannel : function(event) {
            console.log("got receive channel");
            // Call back... "this" is not this
            com.knovz.whiteboard.remote.sendChannel = event.channel;
            com.knovz.whiteboard.remote.sendChannel.onmessage =
com.knovz.whiteboard.remote.handleMessage;
            com.knovz.whiteboard.remote.sendChannel.onopen =
com.knovz.whiteboard.remote.handleReceiveChannelStateChange;
            com.knovz.whiteboard.remote.sendChannel.onclose =
com.knovz.whiteboard.remote.handleReceiveChannelStateChange;
      },
      handleReceiveChannelStateChange : function() {
            var readyState =
com.knovz.whiteboard.remote.sendChannel.readyState;
            console.log('Receive channel state is: ' + readyState);
            // handle button states, etc...
            com.knovz.whiteboard.remote.enableMessageInterface(readyState ==
"open");
      },
      enableMessageInterface : function(shouldEnable) {
            if (shouldEnable) {
                  // TODO hadle needed Interface changes
                  this.isReadyCom = true;
                  // TODO maybe not the place, close Socket!!
                  this.disconnectIO();
            } else {
                  // TODO handle needed Interface changes
            }
      },
      doCall : function() {
            console.log("do call");
            var constraints = {'MozDontOfferDataChannel': true};
            if (webrtcDetectedBrowser === 'chrome') {
                  // Doesn't seem to be necessary
                  for (var prop in constraints) {
                        if (prop.indexOf('Moz') !== -1) {
                              delete constraints[prop];
                        }
                  }
            }
            trace('Send offer to peer');
            this.peerCom.createOffer(this.setLocalAndSendMessage,
this.errorOnCreateOffer, constraints);
      },
      errorOnCreateOffer : function(error) {
            console.log('Error creating Offer: ' + error);
      },
      doAnswer : function() {
            console.log('Sending answer to peer.');
      this.peerCom.createAnswer(this.setLocalAndSendMessage,
this.errorOnCreateAnswer, null);
      },
```

```
	errorOnCreateAnswer : function(error) {
		console.log('Error creating Offer: ' + error);
	},
	setLocalAndSendMessage : function (sessionDescription) {
		// Call back... "this" is window

	com.knovz.whiteboard.remote.peerCom.setLocalDescription(sessionDescrip
tion);

	com.knovz.whiteboard.remote.sendSocketMessage(sessionDescription);
	},
	closeDataChannels : function() {
		// TODO
	},
	sendData: function(data) {
		try {
			this.sendChannel.send(data);
			console.log("Sent Data: " + data);
		} catch(e) {
			console.log("Error sending: " + data);
		}

	},
	handleMessage : function(event) {
		console.log(event);
		var message = JSON.parse(event.data);
		// TODO handle errors
		if(message.type == "chat") {
			com.knovz.whiteboard.remote.chat.value =
com.knovz.whiteboard.remote.chat.value + ">>" + message.message + "\n";
		} else if (message.type == "whiteboard") {
			com.knovz.whiteboard[message.function](message.params,
true, true);
		}
	},
	sendChatMessage : function(message) {
		this.sendData('{"type" : "chat", "message" : "' + message +
'"}');
		this.chat.value = this.chat.value + message + "\n";
	}
}

// Override some whiteboard functions to add remote functionality if there's
comm
com.knovz.whiteboard.setColor = function(color, isRemote) {
	this.color = color;
	if(this.remote.isReadyCom && !isRemote) {
		console.log("Send to remote");
		this.remote.sendData('{"type" : "whiteboard", "function" :
"setColor", "params" : "' + color + '"}');
	}
}

com.knovz.whiteboard.clear = function(none, isRemote) {
	this.context.clearRect(0,0,this.canvas.width,this.canvas.height)
	if(this.remote.isReadyCom && !isRemote) {
		console.log("Send to remote");
		this.remote.sendData('{"type" : "whiteboard", "function" :
"clear", "params" : ""}');
```

```
	}
}

com.knovz.whiteboard.drawingStart = function(obj, event, isRemote) {
	var pos;
	if(isRemote) {
		// obj is the pos
		pos = obj;
	} else {
		// obj and event are just that
		pos = this.events.getPoint(obj, event);
	}

	this.context.strokeStyle = this.color;
	this.context.lineWidth = this.lineWidth;
	this.context.beginPath();
	this.context.moveTo(pos.x, pos.y);
	this.isDrawing = true;

	if(this.remote.isReadyCom && !isRemote) {
		// Send
		this.remote.sendData('{"type" : "whiteboard", "function" :
"drawingStart", "params" : ' + JSON.stringify(pos) + '}');
	}
}

com.knovz.whiteboard.drawingLine = function(obj, event, isRemote) {
	if(this.isDrawing) {
		var pos;
		if(isRemote) {
			// obj is the pos
			pos = obj;
		} else {
			// obj and event are just that
			pos = this.events.getPoint(obj, event);
		}
		this.context.lineTo(pos.x,pos.y);
		this.context.stroke();

		if(this.remote.isReadyCom && !isRemote) {
			// Send
			this.remote.sendData('{"type" : "whiteboard", "function"
: "drawingLine", "params" : ' + JSON.stringify(pos) + '}');
		}
	}
}

com.knovz.whiteboard.drawingEnd = function(obj, event, isRemote) {
	if(this.isDrawing) {
		var pos;
		if(isRemote) {
			// obj is the pos
			pos = obj;
		} else {
			// obj and event are just that
			pos = this.events.getPoint(obj, event);

		}
```

```
            // Drawing Line is sent twice
            this.drawingLine(obj, event, isRemote);
            this.isDrawing = false;

            if(this.remote.isReadyCom && !isRemote) {
                  // Send
                  this.remote.sendData('{"type" : "whiteboard", "function"
: "drawingEnd", "params" : ' + JSON.stringify(pos) + '}');
            }
      }
}

// Launch socket to allow start of signaling
com.knovz.whiteboard.remote.socketInit();
```

Videoplayer.js

Javascript file holding the video functionality. Only local.

```
/*!
* com.knovz.videoplayer v1.0.0
* Licensed under MIT
*/

// Check requisites
if (typeof jQuery === "undefined") { throw new Error("com.knovz.whiteboard
requires jQuery") }

// Namespacing
var com = com || {};
com.knovz = com.knovz || {};

// videoplayer
com.knovz.videoplayer = {
      options : {},
      defaultOptions : {
            texts : {
                  errorMsg : "ERROR"
            }
      },
      video : {},
      videoplayer : {},
      isLoaded : false,
      init : function(options) {
            // Options must be an object containing a canvas id
            if (!((typeof options === "object") && (typeof options.playerId
=== "string") && (options.playerId.length > 0))) { throw new
Error("com.knovz.videoplayer.init requires options in json format, with at
least a non empty string playerId defining the video element to use.") }
            // Combine parameter options with defaultOptions, into options
            this.options = $.extend(true,{},this.defaultOptions,options);
            // Get canvas and 2d context
            this.videoplayer =
document.getElementById(this.options.playerId);
            return this;
      },
      load : function(video) {
```

```
            // video must be an object, containing src, and optionally
poster
            if (!((typeof video === "object") && (typeof video.src ===
"string") && (video.src.length > 0))) { throw new Error("video must be an
object, containing src, and optionally poster.") }
            this.videoplayer.src = video.src;
            if((typeof video.poster === "string") && (video.poster.length >
0)) {
                  this.videoplayer.poster = video.poster;
            }
            // TODO test whether load is necessary.
            // -> load is necessary to change sources programmatically, to
stop all pending operations on previous src,
            //        and start anew with current src
            // With or without load(), duration is not available until first
play() if preload none
            //  and appears to be available if "metadata"
            this.videoplayer.load();
            // Maybe force preload=metadata here (well, before src)
            // this.videoplayer.preload = "metadata";
            this.isLoaded = true;
      },
      setPlayerSize : function(w,h) {
            this.videoplayer.width = w;
            this.videoplayer.height = h;
      },
      togglePlay() {
            // TODO manage interface
            if(!this.isLoaded) {return};
            if(this.videoplayer.paused) {
                  this.videoplayer.play();
            } else {
                  this.videoplayer.pause();
            }
      },
      moveTo : function(secs) {
            // TODO should check video loaded?
            // TODO check in iPad, for video is not loaded until "play"
            if(!this.isLoaded) {return};
            if(this.videoplayer.duration) {
                  if (secs > this.videoplayer.duration) {
                        secs = this.videoplayer.duration;
                  } else if (secs < 0) {
                        secs = 0;
                  }
                  this.videoplayer.currentTime = secs;
            }
      },
      move : function(secs) {
            this.moveTo(this.videoplayer.currentTime + secs);
      },
      framestep : 0.040, // 40ms, 25fps
      moveStepFwd : function() {
            if(!this.videoplayer.paused) {
                  this.togglePlay();
            }
            this.move(this.framestep);
      },
      moveStepBack : function() {
```

```
            if(!this.videoplayer.paused) {
                  this.togglePlay();
            }
            this.move(-1 * this.framestep);
      },
};
```

Videoplayer_remote.js

Javascript file holding the remote video playback functionality. Builds on videoplayer.js, and overwrites some of it's functionality. Relies on whiteboard_remote.js for remote functionality.

```
/*!
* com.knovz.videoplayer.remote v1.0.0
* This is an add-in for com.knovz.videoplayer.
* Relies on com.knovz.whiteboard.remote for remote comunication
* Licensed under MIT
*/

// Namespacing
var com = com || {};
com.knovz = com.knovz || {};

// Check Videoplayer
if (typeof com.knovz.videoplayer === "undefined") { throw new
Error("com.knovz.videoplayer.remote requires com.knovz.videoplayer") }

// Whiteboard remote is used for communication
// TODO separate com part. Too coupled!!!!
if (typeof com.knovz.whiteboard.remote === "undefined") { throw new
Error("com.knovz.videoplayer.remote requires com.knovz.whiteboard.remote") }

// This is so ugly, it makes me weep
com.knovz.videoplayer.remote = com.knovz.whiteboard.remote;

// Override whiteboard.remote.handleMessage to add video
com.knovz.whiteboard.remote.handleMessage = function(event) {
      console.log(event);
      var message = JSON.parse(event.data);
      // TODO handle errors
      if(message.type == "chat") {
            com.knovz.whiteboard.remote.chat.value =
com.knovz.whiteboard.remote.chat.value + ">>" + message.message + "\n";
      } else if (message.type == "whiteboard") {
            com.knovz.whiteboard[message.function](message.params, true,
true);
      } else if (message.type == "videoplayer") {
            com.knovz.videoplayer[message.function](message.params, true,
true);
      }
}

// Override some videoplayer functions to add remote functionality if there's
comm
com.knovz.videoplayer.togglePlay = function(none, isRemote) {
```

```
	console.log("Override");
	if(!this.isLoaded) {return};
	if(this.videoplayer.paused) {
		this.videoplayer.play();
	} else {
		this.videoplayer.pause();
	}

	if(this.remote.isReadyCom && !isRemote) {
		console.log("Send to remote");
		this.remote.sendData('{"type" : "videoplayer", "function" :
"togglePlay", "params" : ""}');
		// There will be a delay. We can adjust with currentTime
		// Only when paused
		if (this.videoplayer.paused) {
			var current = this.videoplayer.currentTime;
			this.remote.sendData('{"type" : "videoplayer", "function"
: "moveTo", "params" : ' + current + '}');
		}
	}
}

// All move orders are togglePlay and moveTo compounds
com.knovz.videoplayer.moveTo = function(secs, isRemote) {
	console.log("Override");
	if(!this.isLoaded) {return};
	if(this.videoplayer.duration) {
		if (secs > this.videoplayer.duration) {
			secs = this.videoplayer.duration;
		} else if (secs < 0) {
			secs = 0;
		}
		this.videoplayer.currentTime = secs;
		if(this.remote.isReadyCom && !isRemote) {
			console.log("Send to remote");
			this.remote.sendData('{"type" : "videoplayer", "function"
: "moveTo", "params" : ' + secs + '}');
		}
	}

}
```