

A WEBRTC BASED PLATFORM FOR SYNCHRONOUS  
ONLINE COLLABORATION AND SCREEN CASTING

by

NIKOLAOS PINIKAS

Master in Informatics and Multimedia, Department of Informatics Engineering,  
School of Applied Technology, Technological Educational Institute of Crete, 2016

A THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN INFORMATICS & MULTIMEDIA

DEPARTMENT OF INFORMATICS ENGINEERING

SCHOOL OF APPLIED TECHNOLOGY

TECHNOLOGICAL EDUCATIONAL INSTITUTE OF CRETE

2016

Approved by:

Assistant Professor  
Dr. Spyros Panagiotakis

# Copyright

NIKOLAOS PINIKAS

2016



This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/>.



All included code is licensed under the MIT License.

*“WebRTC is not a standard. It’s a movement.”*

-Tsahi Levent-Levi

## Abstract

WebRTC is a project that was released by Google in 2011 to allow browser-to-browser communication. It includes voice, video and data without the use of plugins. The mission of WebRTC according to Google is to enable rich, high quality, RTC applications to be developed for the browser, mobile platforms, and Internet of Things (IoT) devices, and allow them all to communicate via a common set of protocols.

In this thesis we employ the capabilities of the WebRTC APIs to implement a platform for synchronous online collaboration, screen casting and simultaneous multimedia communication by utilizing the WebRTC data and media streams. Collaborative software is defined as “a software that supports intentional group processes”. Collaborative solutions include a wide range of tools. On the Web these tools can be part of what is known as a “Client-Portal” and can include discussions, white boards, media and file exchanging etc.

Moving from the world of client-server architecture to the peer-to-peer world the ideas of online collaboration can be applied to offer more immediate synchronous communication without the need of a centralized system.

The APIs that will be mainly used are those provided by WebRTC, the Screen Capturing API, the Media Recording API and other APIs as defined in the corresponding W3C drafts and in the degree they are implemented in modern browsers. Adding screen casting in an online collaboration system can provide useful features such as marking things on the screen, providing insight on what to do next, simultaneous document editing, creating and checking presentations etc. In this thesis we develop a synchronous collaboration platform using only modern web technologies and propose a communication protocol that makes it possible for peers to exchange collaboration data in a real-time communication environment.

*Key words:* WebRTC, HTML5, Whiteboard, Online collaboration, Screen casting, Collaborative browsing

## Περίληψη

Το WebRTC είναι ένα έργο που κυκλοφόρησε από την Google το 2011, το οποίο επιτρέπει στα πρόγραμμα περιήγησης (browsers) να επικοινωνούν μεταξύ τους χρησιμοποιώντας φωνή, βίντεο και δεδομένα χωρίς τη χρήση πρόσθετων προγραμμάτων (plugins). Η αποστολή του WebRTC σύμφωνα με την Google είναι να επιτρέψει την ανάπτυξη υψηλής ποιότητας εφαρμογών επικοινωνίας πραγματικού χρόνου (RTC) για προγράμματα περιήγησης, κινητές πλατφόρμες, και το Internet of Things, μέσα από ένα σύνολο κοινών πρωτοκόλλων.

Σε αυτή η εργασία χρησιμοποιώντας κυρίως τις δυνατότητες του WebRTC API, υλοποιούμε μια πλατφόρμα για online συνεργασία, διαμοιρασμό οθονών και μεταφοράς πολυμέσων χρησιμοποιώντας τις δυνατότητες του WebRTC. Λογισμικά και πλατφόρμες συνεργασίας ορίζονται αυτές που μπορούν να υποστηρίξουν ομαδικές διεργασίες και σήμερα περιλαμβάνουν ένα ευρύ φάσμα εργαλείων που περιλαμβάνουν συζητήσεις, ανταλλαγή πολυμέσων, οθονών, αρχείων κλπ.

Μεταβαίνοντας από την αρχιτεκτονική client-server στον κόσμο της αρχιτεκτονικής peer-to-peer, αυτές οι ιδέες της online συνεργασίας μπορούν να υλοποιηθούν πλέον χωρίς την ανάγκη ύπαρξης ενός κεντρικού συστήματος.

Τα API που χρησιμοποιούμε σε αυτήν την εργασία είναι αυτά που παρέχονται από το WebRTC, το API για καταγραφή οθονών (screen capturing), για εγγραφή μέσων (media recording) και άλλα API όπως ορίζονται από το W3C και στο βαθμό που έχουν υλοποιηθεί στους σύγχρονους browsers. Προσθέτοντας δυνατότητα καταγραφής της οθόνης ή μέρους αυτής και στη συνέχεια διαμοιρασμού της στους συμμετέχοντες στο συνεργατικό περιβάλλον, γίνεται εφικτή η υλοποίηση μιας σειράς λειτουργιών όπως σημείωση σε μέρος της οθόνης, παροχή οδηγιών και τεχνικής υποστήριξης, ταυτόχρονη επεξεργασία εγγράφων, δημιουργία και έλεγχος παρουσιάσεων κλπ. Σε αυτή την εργασία αναπτύσσουμε μια συνεργατική εφαρμογή χρησιμοποιώντας αυτές τις σύγχρονες τεχνολογίες του Web, και προτείνουμε ένα πρωτόκολλο για ανταλλαγή δεδομένων σε πραγματικό χρόνο σε ένα συνεργατικό περιβάλλον.

*Λέξεις κλειδιά:* WebRTC, HTML5, Ασπροπίνακας, Σύγχρονη online συνεργασία, Screen casting, Συνεργατική πλοήγηση στο web

# Table of Contents

<b>Copyright</b> .....	2
<b>Abstract</b> .....	4
<b>Περίληψη</b> .....	5
<b>Table of Contents</b> .....	6
<b>List of Figures &amp; Screenshots</b> .....	8
<b>List of Tables</b> .....	9
<b>List of Listings</b> .....	9
<b>Abbreviation Index</b> .....	11
<b>Acknowledgments</b> .....	13
<b>Introduction</b> .....	14
<b>Chapter 1. Online Collaboration &amp; P2P Media Streaming</b> .....	16
1.1 Online Collaboration.....	16
1.2 Online Collaboration Related Work.....	18
1.3 Co-browsing.....	18
1.4 Co-browsing Related Work.....	19
1.5 Media Streaming.....	20
<b>Chapter 2. Underlying Technologies</b> .....	21
2.1 WebSocket & HTTP/2.....	21
2.2 WebRTC.....	23
2.2.1 WebRTC Use Cases.....	24
2.2.2 WebRTC APIs.....	26
WebRTC Media Stream API.....	26
WebRTC Peer Connection API.....	28
WebRTC Data Channel.....	32
WebRTC Screen Casting.....	34
Screen Casting Security Issues.....	36
WebRTC Interoperability.....	38
2.2.3 WebRTC Signaling.....	40
Node.js/Socket.io for WebRTC Signaling.....	41
2.2.4 WebRTC Network Protocols.....	42

SDP .....	42
STUN .....	43
TURN.....	44
ICE .....	45
2.2.5 WebRTC Codecs .....	47
2.3 HTML5 .....	48
2.3.1 HTML5 APIs .....	49
File API and the Blob Interface .....	49
Stream Capture from DOM Elements API .....	50
Media Recording API .....	51
2.3.2 jQuery & jQueryUI.....	52
2.4 Data Chanel Compression.....	53
<b>Chapter 3. Communication Protocol.....</b>	<b>55</b>
3.1 Use of the native WebRTC send/receive functions .....	56
3.2 Sending data with sendDataAction .....	57
3.3 Strings defining actions.....	60
3.4 The handleMessage function .....	63
3.5 Example of Expandability.....	64
<b>Chapter 4. Implementation .....</b>	<b>66</b>
4.1 Infrastructure.....	66
4.2 Implementation of the Signaling Server .....	68
4.3 The Client Application.....	70
4.4 The Interface .....	71
<b>Chapter 5. Benchmarking .....</b>	<b>82</b>
5.1 Compression Efficiency.....	82
5.2 CPU Consumption .....	85
<b>Chapter 6. Conclusions &amp; Future Work.....</b>	<b>87</b>
<b>Chapter 7. References .....</b>	<b>89</b>

## List of Figures & Screenshots

<b>Figure 1.1</b>	A possible scenario of non-interactive co-browsing .....	19
<b>Figure 2.1</b>	Traditional client server bidirectional communication employed in WebSocket ..	21
<b>Figure 2.2</b>	Server push used in HTTP/2 .....	21
<b>Figure 2.3</b>	Asking for the user's permission to use a device or start capturing the screen...	217
<b>Figure 2.4</b>	RTCPeerConnection structure .....	217
<b>Figure 2.5</b>	Screen capturing modes .....	35
<b>Figure 2.6</b>	Obscuring information in application capturing mode.....	36
<b>Figure 2.7</b>	Notifying the user that screen sharing is active in Firefox.....	37
<b>Figure 2.8</b>	Using screen sharing for launching CSRF attacks .....	378
<b>Figure 2.9</b>	WebRTC Signaling Architecture .....	40
<b>Figure 2.10</b>	Connection using a STUN Server .....	44
<b>Figure 2.11</b>	Using a TURN Server to relay data .....	44
<b>Figure 2.12</b>	Sample list of local and remote ICE candidates gathered by Mozilla Firefox..	446
<b>Figure 2.13</b>	List of local and remote ICE candidates gatered by Mozilla Firefox.....	46
<b>Figure 2.14</b>	Using WebRTC to reveal user's IP information.....	46
<b>Figure 2.15</b>	WebRTC signaling complete process .....	47
<b>Figure 2.16</b>	Usage of jQuery for websites, 15 Feb 2015 to 15 Feb 2016 .....	53
<b>Figure 2.17</b>	Data channel compression schematic.....	53
<b>Figure 2.18</b>	LZW Algorithm Flowchart .....	54
<b>Figure 3.1</b>	Communication Model.....	56
<b>Figure 3.2</b>	System components and sample incoming messages.....	631
<b>Figure 3.3</b>	Converting messages into function calls .....	63
<b>Figure 3.4</b>	Poke message results.....	635
<b>Figure 4.1</b>	Application data per type .....	66
<b>Figure 4.2</b>	The BeagleBone Black Single Board Computer.....	67
<b>Figure 4.3</b>	The application interface.....	71
<b>Figure 4.4</b>	Sample stream thumbnails with available actions. From left to right: window, webcam and a local video file .....	72
<b>Figure 4.5</b>	A screen capturing session with the PowerPoint window mazimized .....	73
<b>Figure 4.6</b>	The whiteboard toolbar .....	73
<b>Figure 4.7</b>	Example system messages .....	75
<b>Figure 4.8</b>	Users exchanging files .....	76
<b>Figure 4.9</b>	Users sketching on a PDF document.....	76



<b>Figure 4.10</b> Recording and then playing a WebM file locally.....	77
<b>Figure 4.11</b> Selecting to embed incoming chat messages on the vide ostream .....	78
<b>Figure 4.12</b> Adding text annotation on a shared webm video .....	79
<b>Figure 4.13</b> Resulting sketch .....	80
<b>Figure 5.1</b> 103KB of uncompressed sketch data .....	82
<b>Figure 5.2</b> Compression Efficiency .....	823
<b>Figure 5.3</b> Comparission between compression and no compression on a LAN .....	824
<b>Figure 5.4</b> Comparission of frame rate during simple streaming and during sketching.....	826

## List of Tables

<b>Table 2.1</b> Comparission of WebSocket abd HTTP/2.....	22
<b>Table 2.2</b> RTCPeerConnection main methods.....	30
<b>Table 2.3</b> Comparission between WebSOcket and the WebRTC Data Channel.....	302
<b>Table 2.4</b> List of the most important members of the RTCDataChannelInit collection .....	33
<b>Table 2.5</b> Example API differences accross Chrome and Firefox.....	38
<b>Table 3.1</b> Sample communcation prefixes.....	62
<b>Table 4.1</b> BeagleBoard Black Specifications.....	67
<b>Table 4.2</b> List of applicacion files .....	70
<b>Table 5.1</b> Compression efficiency .....	83
<b>Table 5.2</b> Data transfer rates without compression.....	84
<b>Table 5.3</b> Firefox configuration information indicating hardware accelaration using the Windows Direct3D.....	85

## List of Listings

<b>Listing 2.1</b> Syntax of the getUserMedia method .....	27
<b>Listing 2.2</b> getUserMedia constraint to access the rear camera of the device .....	27
<b>Listing 2.3</b> Sample returned data from getSupportedConstraints on a laptop computer .....	28
<b>Listing 2.4</b> Creating an RTCPeerConnection object.....	29
<b>Listing 2.5</b> Setting the local description using the createOffer method.....	30
<b>Listing 2.6</b> Setting the remote description and adding ICE Candidates .....	30
<b>Listing 2.7</b> Defining a callback function for when a new RTCPeerConnection negotiation is needed .....	31
<b>Listing 2.8</b> Creating an unreliable data channel.....	33

<b>Listing 2.9</b> Creating a reliable data channel .....	33
<b>Listing 2.10</b> MediaStreamConstraints object for Application capturing in Firefox .....	34
<b>Listing 2.11</b> Malicious HTML file for a possible CSRF attack through screen sharing .....	38
<b>Listing 2.12</b> Detecting the browser and intercepting WebRTC API accordingly .....	39
<b>Listing 2.13</b> Mitigating the lack of MediaDevices interface in Adapter.js .....	40
<b>Listing 2.14</b> Socket.io on the server side .....	41
<b>Listing 2.15</b> Socket.io on the client side .....	42
<b>Listing 2.16</b> Excerpt of SDP message .....	43
<b>Listing 2.17</b> Creating an RTCPeerConnection object and telling it which STUN and TURN servers to use .....	45
<b>Listing 2.18</b> HTML containing obtrusive JavaScript .....	52
<b>Listing 2.19</b> Same HTML code with JavaScript now removed .....	52
<b>Listing 2.20</b> Unobtrusive JavaScript located in a separate .js file .....	52
<b>Listing 2.21</b> Unobtrusive JavaScript located in a separate .js file using jQuery .....	52
<b>Listing 3.1</b> Usage of the RTCDataChannel send() method and the onmessage callback .....	57
<b>Listing 3.2</b> Definition of the interface in Web IDL .....	57
<b>Listing 3.3</b> The sendDataAction function and an example call .....	59
<b>Listing 3.4</b> Sending files over the data channel .....	60
<b>Listing 3.5</b> Sample Sketching Message .....	62
<b>Listing 3.6</b> Using the handleMessage function .....	64
<b>Listing 3.7</b> Calling the sendDataAction to send a string and the handling function .....	65
<b>Listing 4.1</b> Starting the Node.js server script using Forever .....	68
<b>Listing 4.2</b> Creating an SSL server in Node.js .....	68
<b>Listing 4.3</b> Creating a socket.io room on the signaling server .....	69
<b>Listing 4.4</b> Broadcasting messages from the signaling server .....	69
<b>Listing 4.5</b> Using the capturestream method to capture a stream from a video element .....	72
<b>Listing 4.6</b> Constructing the chat message .....	74
<b>Listing 4.7</b> The Urmsg function .....	75
<b>Listing 4.8</b> The file function .....	76
<b>Listing 4.9</b> Sample sketching actions JSON object .....	79
<b>Listing 4.10</b> Unicode characters resulting from compressing a string to UTF16 LZW .....	80
<b>Listing 4.11</b> The sktch function .....	81

## Abbreviation Index

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
B2B	Business to Business
BLOB	Binary Large Object
CDN	Content Delivery Network
CPU	Central Processing Unit
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheets
DOM	Document Object Model
GPU	Graphics Processing Unit
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ICE	Interactive Connectivity Establishment
IETF	Internet Engineering Task Force
ILBC	Internet Low Bitrate Codec
IOT	Internet of Things
IP	Internet Protocol
ISAC	Internet Speech Audio Codec
ITU	International Telecommunications Union
JS	JavaScript
JSEP	JavaScript Session Establishment Protocol
JSON	JavaScript Object Notation
LAN	Local Area Network
LZW	Lempel-Ziv-Welch Compression
MCU	Multipoint Control Unit
NAT	Network Address Translation
P2P	Peer to Peer
P2PTV	Peer to Peer Television

PCMA	Pulse Code Modulation A-law
PCMU	Pulse Code Modulation $\mu$ -law
PEM	Privacy Enhanced Mail
PNG	Portable Network Graphics
RFC	Request for Comments
RTC	Real Time Communication
SCTP	Stream Control Transmission Protocol
SDK	Software Development Kit
SDP	Session Description Protocol
SIP	Session Initiation Protocol
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
UX	User Experience
W3C	World Wide Web Consortium
WebRTC	Web Real Time Communication
XMPP	Extensible Messaging and Presence Protocol

## **Acknowledgments**

I would like to thank my thesis supervisor, Assistant Professor Dr. Spyros Panagiotakis for his kind support and encouragement. Dr. Athanasios Malamos from the TEI of Crete Multimedia Content Laboratory offered some valuable feedback and suggestions. My gratitude goes to the Mozilla developers for being first in implementing all the HTML5 and WebRTC W3C API drafts that are used in the prototype application developed for the needs of this thesis. They saved me a lot of trouble and effort.

I would also like to thank all my friends at Crete who were catalytic in providing the essential incentive to complete this work. You guys are great! Finally a special thank you goes to Lambros Frantzeskakis for his “psychological” support.

*Heraklion, June 2016*

## Introduction

Since 1990, when the World Wide Web was born at CERN, the Web has evolved from a presentation layer powered by a simple markup language (HTML) to a full-scale application environment. In the early 1990s, most Web sites were based on a series of complete HTML pages (one static HTML file for each page). The user would click on a hyperlink and a new HTML page would be loaded from the server although only some minor information had changed, placing additional load on the server and used excessive bandwidth. This inefficient process was reflected in the UX of the Web making it a clumsy “static pull-request media” not better than teletext television services of the time.

The first step to offer users richer web applications was made with the introduction of asynchronous communication on the Web. Ajax was one of the first such technologies to allow asynchronous web communication. With Ajax, web applications could now send data to and retrieve from a server in the background without interfering with the display and behavior of the web page. An application of Ajax used daily by most users of the Web is the auto complete feature of search engines such as Google. When a user start typing in the search box, Ajax sends the typed letters to the web server and the server returns a list of suggestions which are then displayed under the search box.

Because of the asynchronous nature of Ajax, each chunk of data that is sent or received by the client occurs in a connection established specifically for that event. This creates a requirement that for every action, the client should poll the server, instead of listening, which again incurs significant overhead, and in turn to several times higher latency with. The next technological leap was to offer not only asynchronous communication but also persistent connections [1]. This was made possible with the introduction of WebSocket. WebSocket allows not only asynchronous communication but also a persistent connection allowing for full-duplex communication over a single TCP connection.

Another technology that was made available recently is the allowance for browser to browser (P2P) communication. Until now the web was based on the client-server model meaning that all communication between two or more users had to be relayed through a web server. This paradigm was changed with the introduction of WebRTC which is the main focus of this thesis. WebRTC is an API definition

designed to allow browser-to-browser communication without the need of plugins, enabling various kinds of real time communication such as audio, video and data. With browser-to-browser communication users can now communicate in a peer-to-peer fashion and send voice, video and any other message they see fit, eliminating the need for a server. This way, web developers can build web services that require less processing and bandwidth in their backend [2].

Finally what powers the Web today and enables web applications to use the technologies mentioned above is HTML5. To understand the importance of HTML5 we can observe how we have now come to a point where taking HTML5 web apps and wrapping them as native apps (e.g Windows or Android apps) is a common practice. One such technology is Apache Cordova which allows the use of HTML5, CSS3, and JavaScript for cross-platform development. Applications developed with Cordova execute within wrappers targeted to each platform, and rely on standards-compliant API bindings to access each device's hardware capabilities [3].

In this thesis we present the latest developments on WebRTC. We have developed a prototype application that includes example implementations of most of the currently available WebRTC technologies such as screen sharing, media streaming, multiple streams, media recording, canvas integration etc. In chapter 1 a brief literature review on online collaboration/co-browsing and media streaming is conducted. In chapter 2 an in depth review of the technologies used in the development of our application is presented. In chapter 3 the specifications of our suggested communications protocol is discussed. In chapter 4 we present our prototype application and give implementation details. Finally in chapter 5 we discuss benchmarking and performance measurements of the application.

# Chapter 1. Online Collaboration & P2P Media

## Streaming

The main focus of this thesis is on online collaboration, media streaming and co-browsing. In this chapter we present current techniques, trends and related work on these technologies.

### 1.1 Online Collaboration

Collaboration can be defined as the common effort of a group of people to create something. Tools that aid collaboration were around long before computers - whiteboards, flipcharts or even a piece of paper can be used to support collaboration [4]. Computer and the Web revolutionized the way people work together in groups. In the 80s the term “groupware” was coined by C. A. Ellis who defined it as “computer-based system that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment” [5]. Popular groupware software packages included Lotus Notes and Microsoft Exchange.

The Web opened a new window for the development of collaboration software. With Web 2.0 came a plethora of cloud hosted Internet-based apps that enabled more collaboration, formation of online communities, and other means of interaction. Today online collaboration tools can be classified in two categories [6]:

- ◆ **Asynchronous collaboration tools.** These tools enable participants to collaborate on work at different times and different locations. These tools are useful for collaborating over time and providing resources and information that are accessible at any time. Viewing the revision history allows participants to see who has contributed, when they have contributed, and what they have contributed. Plus, the comments allow participants to agree, rebut, or explain changes needed in the work.
- ◆ **Synchronous collaboration tools.** These tools enable participants to collaborate in real-time, whether in the same location or in different places. The key point of synchronous tools is that the technology lets the communicators work together at the same time.



The emphasis of this thesis is on synchronous online collaboration since these kinds of tools are now made possible with real-time technologies such as WebRTC. Synchronous collaboration can have many advantages likes [7]:

- Immediate response and feedback.
- Video/web conferencing allow for body language and tone of voice.
- Increased motivation and engagement with course concepts.
- Increased social presence

Disadvantages of synchronous collaboration include:

- Lack of reflection between collaborators.
- If technology fails the collaboration session not possible.
- Large time commitment for collaborators.
- Difficult for one to many communication.

Synchronous collaboration includes whiteboards, video and audio communication, text chat and screen sharing. Whiteboarding in particular is a teaching and collaboration practice in which participants use a whiteboard area to draw or write concepts, charts, maps, tables, diagrams, equations etc. Smith et al. in [8] conducted a literature review on interactive whiteboard and found among other things that they are particularly effective in education and virtual classrooms allowing teachers to use teaching time to discuss student-generated ideas rather than merely presenting information and summarized the benefits of interactive whiteboards as follows: flexibility and multiple facets, effectiveness in multimedia use; support for the lesson plan; diverse resources; development of information and communication technology skills; and more interaction and student participation in classes. Interactive whiteboards engage students with their peers in a collaborative learning community and it allows for “more than one teacher” in a classroom by allowing students with whiteboards to become teachers as well [9]. This enhances motivation, participation and cooperation. Educational whiteboards are proved to be an effective learning tool for people of all ages. For example Akbaş et al. in [10] have evaluated a whiteboard-based system that trained older people to use automatic teller machines.

## **1.2 Online Collaboration Related Work**

A number of synchronous online collaboration platforms have been proposed or implemented commercially. As early as 1991, Abdel-Wahab et al. in [11] proposed a distributed system that allowed the sharing of X Window applications synchronously among a group of remotely located users. Jara C. in [12] proposed a web learning system which combines synchronous collaborate learning with 3D virtual laboratories. They intergraded their framework in the popular EJS physics platform, allowing users to collaborate using the WebGL platform. Andrioti Z. in [13] combined WebRTC and the Evie-m platform [14] to create an online collaborative educational virtual environment for teaching mathematics. In the field of whiteboarding which is one of the most popular applications of synchronous online collaboration a study by Metz et al. in [15] designed a collaborative whiteboard and evaluated it by assigning tasks to a group of users and collecting data from user interactions and chat communication. They showed that whiteboard can be an effective collaboration tool. Interestingly they observed that the collective consciousness of the group of users is created through off-task interactions. It can be deduced that this ability to have “off-task interaction” is one of the reasons that video communication significantly improves collaboration efficiency and is one of the advantages of synchronous collaboration. Today many online whiteboards are commercially available on the web.

## **1.3 Co-browsing**

Collaborative Web Browsing (co-browsing) is another form of online collaboration in which two or more user navigate the World Wide Web together by sharing a synchronized common view on a web page as well as sharing interactions, such as mouse movements, text highlighting or mouse clicks, on this web page with each other [16]. For example, a B2B customer having difficulty placing an order could call a customer service representative who could then show the customer how to use the ordering pages as though the customer were using their own mouse and keyboard. Collaborative browsing can include e-mail, fax, regular telephone, and Internet phone contact as part of an interaction. Effectively, collaborative browsing allows a company and a customer to "be on the same page." [17] The ability for consumers to share their screens with agents and navigate the Website together, fill

out forms, or find information can enable businesses to increase revenue and quickly resolve support issues [18].

## 1.4 Co-browsing Related Work

One of the earliest attempts to implement a co-browsing system dates back to 1998 before the advent of broadband internet. In [19] it was attempted to implement internet navigation by requesting and navigating a web page using the telephone line. One more serious attempt for real client-server co-browsing was proposed in [20]. In this US patent it proposed that co-browsing can be achieved by utilizing a server that retrieves content of a page on behalf of a collaboration participant or attendee. Each peer operates or views the content with a browser that is augmented with a collaboration applet. Tags, links, script code and other references that may cause a different page to be accessed or loaded from the current page are transformed or replaced on the server before the page is distributed to the attendees. In particular, events and redirections that may cause the attendee browser to directly navigate to another page are transformed on the server. Pre-determined rules may be applied to prevent some attendees from viewing certain content (e.g., financial or personal data). A page may be further transformed at a client browser, to redirect a hyperlink to the collaboration server or to trap some other event. What is described is essentially a client-server, plugin-based solution and is the basis of the majority of co-browsing solution available today while in this thesis we aspire to propose a peer-to-peer plugin-less implementation.

Non interactive co-browsing can be very simple to implement and can utilize the screen sharing API of WebRTC as shown in the following screenshot:

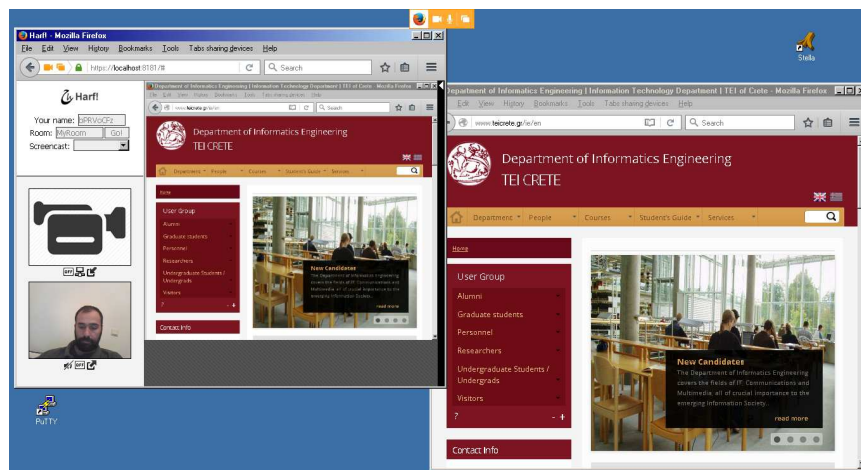


Figure 1.1 A possible scenario of non-interactive co-browsing

In the above scenario a user requests support from an agent. The agent then proceeds to share his/her screen (right window) with the user in an attempt to show the user what to do next while the user watches the screen sharing stream and acts on his own browser accordingly (left window).

The real challenge is implementing an interactive co-browsing session. The obstacles that need to be overcome in plugin-less interactive co-browsing include dealing with cookies, page personalization, login sessions, or requests for authentication while dealing with the strong security measures and confidence requirements provided by both the operating system and the web browser (with most important security limitation being the “Same origin policy” which is discussed in chapter “Screen Casting Security Issues”. In client-server based co-browsing system a solution has been proposed in [16] by enabling the user to control which web application data is propagated and to enforce privacy policies upon private data within a co-browsing session.

## **1.5 Media Streaming**

Media streaming is defined as multimedia that is constantly received by and presented to the end user. According to estimates in 2015 streaming media was accounting for 70% of Internet downstream traffic in North America [21], at the same time regular HTTP traffic accounted for about 6%. In the future with the rise of 4K streaming the percentage of streaming traffic will be increased. Although traditional client-server systems were used initially for delivering media content, researchers and practitioners soon realized that peer-to-peer systems, due to their self-scaling properties, had the potential to improve scalability compared with traditional client-server architectures. Various P2P media streaming systems have been deployed successfully, and corresponding theoretical investigations have been performed on such systems [22]. The term P2PTV refers to peer-to-peer software applications designed to redistribute video streams in real time on a P2P network.

Using WebRTC technology to stream media is still an experimental technology and it is one of the things we will showcase in this thesis.

## Chapter 2. Underlying Technologies

### 2.1 WebSocket & HTTP/2

WebSocket is a technology that makes it possible to open a bidirectional communication session between the user's browser and a server. Before WebSocket, real-time client-server web applications were only possible using the inefficient server polling. Polling is a technique with which the client polls the server at regular intervals and receives the response. However this is obviously not an efficient method because it leads to many connections opening and closing needlessly since real-time data is not always predictable [23]. Using the WebSocket API, a Web application can send messages to a server and receive event-driven responses without having to poll the server for a reply (full-duplex communication) [24]. A connection is established during the initial handshake between the client and the server upgrading in this way the standard HTTP protocol [25].

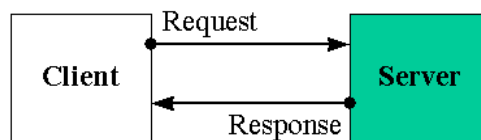


Figure 2.1 Traditional client server bidirectional communication employed in WebSocket

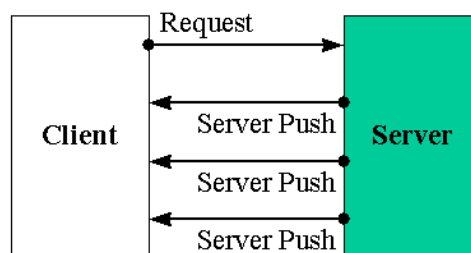


Figure 2.2 Server push used in HTTP/2

The WebSocket protocol was standardized by the IETF as RFC 6455 in 2011 and its API is maintained by W3C. It provides an object and methods that can be used to connect to a server and send a receive data from the connection. The main interface is the WebSocket interface. Many wrapper APIs for the WebSocket API exist, one of the most popular being Socket.IO which is used in this thesis and discussed in a

following chapter. On the server side it becomes obvious that the traditional server stacks are not adequate for the large number of connections. Keeping a large number of connections open at the same time requires an architecture that receives high concurrency at a low performance cost. Such architectures are usually designed around either threading or so called non-blocking IO [26]. Server side implementations include Socket.IO, Websocket-Node for Node.js, Jetty for Java, SuperWebSocket for .NET, Tornado for Python etc. WebSocket is omnipresent in the modern web with applications in social networking and chat, multiplayer games, collaborative applications, online education etc [27].

An alternative to WebSocket is HTTP/2. HTTP/2 is a protocol intended to replace HTTP/1.1 which is used since 1999. It was developed by the IETF HTTP Working Group and is primarily focused on improving the speed to render a webpage. It defines an upgrade handshake and data framing very similar to the WebSocket standard. It is a fully multiplexed binary protocol that uses header compression and allows “server push” [28]. Server Push is where the server pushes a resource directly to the client without the client asking for the resource. HTTP/2 could be used as an alternative to WebSocket. The differences between the two are shown in the following table.

	<b>WebSocket</b>	<b>HTTP/2</b>
Headers	Binary	Binary, compressed
Content	Binary, text	Text, compressed
Direction	Bidirectional	Client to Server, Server push
Multiplexing	Supported (extension)	Supported

*Table 2.1 Comparison of WebSocket and HTTP/2*

An IETF draft has been written for WebSocket over HTTP/2 that describes how WebSocket semantics can be layered onto HTTP/2.0 semantics by defining detailed mapping, replacement of operations and events defined in the WebSocket protocol [29].

## 2.2 WebRTC

WebRTC (Web Real Time Communication) is a technology that allows real-time peer-to-peer communication between browsers without the use of additional plugins. The mission of WebRTC is “to enable rich, high-quality RTC applications to be developed for the browser, mobile platforms, and IoT devices, and allow them all to communicate via a common set of protocols” [30]. WebRTC was open-sourced by Google in 2011 and after that an ongoing work started to standardize the protocols associated with it by IETF and its browser APIs by W3C. Interest and support for WebRTC has been since growing steadily. Today, the most advanced WebRTC implementation is done by Mozilla Firefox and Google Chrome. These browsers are now supporting the majority of the features of WebRTC that are envisioned by the corresponding W3C drafts and proposals [31]. Other platforms that support WebRTC to some extent include the Opera browser, the Android platform and Apple’s iOS platform. Microsoft in its Edge browsers supports another set protocols named ORTC which does not use the SDP for session descriptions but it is planned to be interoperable with WebRTC [32]. It is expected that by 2018 WebRTC will be supported by 4.7 billion mobile devices [33] and 1.5 billion PCs that run WebRTC enabled browsers bringing the total number to over 6.2 billion WebRTC enabled devices.

WebRTC opens the window to a new era of Web innovations that will rely on the web browser for a variety of new activities which were not possible in the past without the need of specialized software or plugins. Video and audio chat, file sharing between peers without the use of an intermediating server, multiplayer games that exchange their data peer-to-peer are just a few of the applications which are made possible by WebRTC. We further discuss all these possibilities in next chapter where we look at some WebRTC use cases.

One of the most important issues with WebRTC is its interoperability. Firstly, the WebRTC web API has not yet fully standardized. As a result web browsers implement slightly different APIs which in turn this has led to Google releasing a shim JavaScript library called adapter.js to insulate WebRTC applications for API changes in the future and across different browsers [34]. We will discuss adapter.js in chapter 2.1.1. Secondly, until recently there hasn’t been an agreement on the set of video codecs which would be used by WebRTC. This held the whole WebRTC

ecosystem back for some time. The two proposed codecs were Google's VP8 and MPEG's H.264.Constrained Baseline. Agreeing on a set of video and audio codecs is important because browsers running WebRTC applications should all support the same set of video and audio codecs because lack of support on the same codec set would break interoperability [35]. It was recently decided that both VP8 and H.264 to be mandatory to implement [36]. Audio codecs which have been decided for audio are Opus and G.711 [37]. We will extensively discuss WebRTC codecs in section 2.2.5.

### 2.2.1 WebRTC Use Cases

As stated in the previous section, it is estimated that over 6.2 billion devices will be WebRTC enabled by 2018. The main question that arises is “What will we do with these devices?” and “What can we develop with WebRTC?”. Currently there are over 200 commercial solutions utilizing WebRTC with new ones released constantly [38]. These applications range from simple online video conferencing to file sharing and torrent sharing to healthcare systems and even distributed CDN system. In this chapter we look at some established examples and innovations that rely on WebRTC technology.

One of the challenges that WebRTC faces is its adoption by the mobile world. Currently mobile versions of Chrome, Firefox and Opera support WebRTC on the Android platform. *Browser* was developed by Ericsson Research and is the first browser that supported WebRTC for the iOS platform [39]. An alternative to web applications, is native mobile applications that utilize the WebRTC Native Code which is offered by Google and Ericsson for Android and iOS. This way, native RTC applications can be developed able to communicate with any WebRTC device. 3rd party SDKs for building native WebRTC applications, such as *easyRTC*, are also available.

The most basic use of WebRTC is in the field of teleconferencing and audio/video communication. Firefox Hello is a feature built-in Mozilla Firefox which enables video and voice calls and text messaging [40]. Zingaya is a click-to-call service for use on websites that allows visitors to video chat with a sales representative/support person etc. eliminating the need for a telephone call [41]. WebRTC has also been used in synchronized software development and collaborative language learning as described in [42] and [43].



There are numerous proposed application of WebRTC in the field of telehealth and telemedicine. Some notable examples include Cola et al. in [44] who propose a video appointment solution that allows doctor and patient to have a video consult instead of a normal visit at the physician office. Vidul et al. in [45] propose a new Emergency Telemedicine Application for emergency care management which uses WebRTC. A WebRTC enabled device is carried within an ambulance to conduct an initial assessment of the patient and later brought to the nearest health center where further treatment is carried under the assistance of specialists whose telepresence is provided by WebRTC enabled devices. Jang-Jaccard in [46] propose WebRTC-based video conferencing system which allows online meetings between remotely located care coordinators and patients at their home while in [47] an efficient session weight load balancing and scheduling methodology to improve network performance for a telehealth care service based on WebRTC is proposed.

Utilizing the data channel for distribution of web content and file sharing peer-to-peer is another area in which WebRTC has a prominent presence. Sharefest and Webtorrent are two examples. Sharefest allows a user to drag and drop a file on a web page. The file is not uploaded anywhere instead the user is given a URL. Using that URL other users can download the file directly from the first user using WebRTC (provided the first user's browser remains open) [48]. WebTorrent is a torrent client developed for the browser. Currently it is only compatible with the WebRTC enabled BitTorrent clients and thus it cannot download data from conventional torrent clients [49]. P2P CDN is another field in which WebRTC can be used. Traditionally CDNs are server-based networks, but when P2P is employed it means that instead of always serving content directly from the CDN to the end users, the end users can share content or blocks of it between them, which reduces the load on the CDN and the bandwidth required on the CDN's side [50]. Notable example of WebRC based CDNs include peerCDN and Peer5. Peer5 is used for distributing video streaming among viewers and can purportedly reach up to 95% server offloading [51]. peerCDN which was acquired by Yahoo in 2013 is another P2P content delivery network that utilizes WebRTC to distribute site resources like images, videos and downloads among site visitors [52].

Using WebRTC for exchanging metadata in web based real-time games has the advantage of reduced server bandwidth, lower latencies and also enables players to

interact via audio and video. Andrioti et al. in [53] have implemented a demo 3D collaborative online game introducing WebRTC over X3DOM technology.

Finally the emerging popularity of WebRTC has led many companies to offer dedicated signaling and hosting services, while other companies offer WebRTC API wrappers. Notable examples include Xirsys, TokBox and peerJS. Xirsys offers STUN and TURN server hosting for WebRTC applications [54], peerJS wraps the browser's WebRTC implementation to provide a complete, configurable, and easy-to-use API [55] and TokBox provides hosted infrastructure, APIs and tools required to deliver enterprise-grade WebRTC capabilities [56].

### **2.2.2 WebRTC APIs**

WebRTC implements three APIs:

- ◆ `MediaStream`
- ◆ `RTCPeerConnection`
- ◆ `RTCDataChannel`

The `MediaStream` API is responsible for capturing streams of media, these streams can be a video taken from the user's web camera, a stream from a canvas of video element or a screen casting session, the `RTCPeerConnection` API is used to communicate these streams between browsers and the `RTCDataChannel` API is used to exchange arbitrary data such as application and game data but also metadata. We will look at these 3 APIs in some more detail in the following paragraphs.

### **WebRTC Media Stream API**

At the heart of the WebRTC API lies the `MediaStream Processing API`, often called the `Media Stream API` or the `Stream API`. This API describes a stream of audio or video data, the methods with working with them, the constraints associated with the type of data, the success and error callbacks when using the data asynchronously, and the events that are fired during the process [57]. Media streams are distinguished between local and remote. The source of a local stream can be the user's web camera or microphone, a screen capturing stream, an HTML5 canvas or a video element. We discuss with more detail in screen capturing and in stream capturing from video and canvas elements in later sections. The source of a non-local `MediaStream` may be a

stream originating over the network, and obtained via the WebRTC PeerConnection API, or a stream created using the Web Audio API `MediaStreamAudioSourceNode`.

Local media streams from the web cam or screen capture are generated by the `MediaDevices.getUserMedia` method which prompts the user for permission to use one video and/or one audio input device such as a camera, a microphone or for permission to start capturing a screen or part of it.

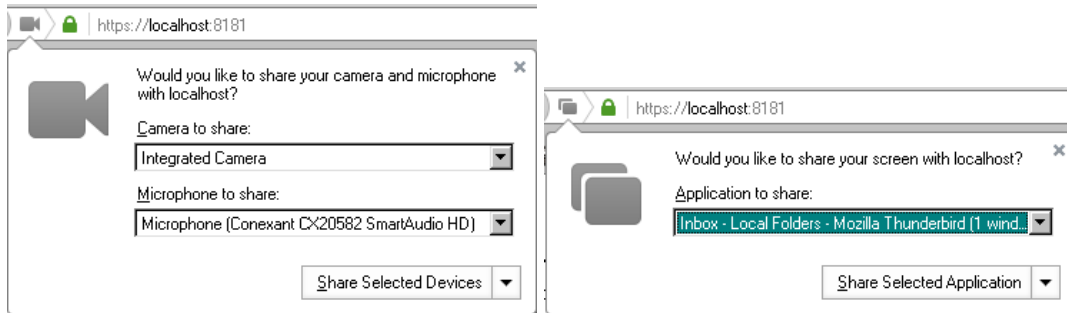


Figure 2.3 Asking for the user's permission to use a device (left) or start capturing part of the screen (right)

If the user provides permission, then the returned Promise (an object is used for deferred and asynchronous computations) is resolved with the resulting `MediaStream` object. If the user denies permission, or media is not available, then the promise is rejected with `PermissionDeniedError` or `NotFoundError` respectively. [58].

The `getUserMedia` method has one `constraints` parameter and two callback functions, one for successful creation of the promise and one for the rejection of the returned promise.

```
navigator.mediaDevices.getUserMedia(constraints)
  .then(successFunction)
  .catch(errorFunction)
```

Listing 2.1 Syntax of the `getUserMedia` method

The *constraints* object contains parameters such as the video resolution, frame rate or which camera to use on devices with more than one camera. For example to enable audio and allow access to the rear camera of a device instead of the front camera the following `constraints` object can be used:

```
{audio:true, video:{facingMode:{exact:"environment"}}}
```

Listing 2.2 `getUserMedia` constraint to access the rear (environment) camera instead of the front camera of the device

Because not all devices support all constraints W3C requires the implementation of a `getSupportedConstraints` method which returns all the supported constraints of the devices as seen in the following listing:

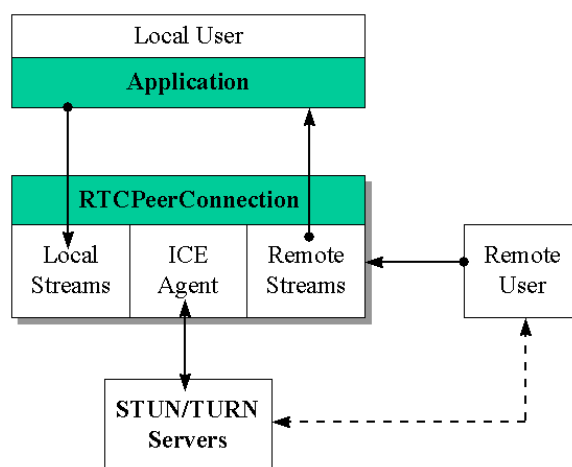
```
Object { browserWindow: true,
        deviceId: true,
        facingMode: true,
        frameRate: true,
        height: true,
        mediaSource: true,
        scrollWithPage: true,
        width: true }
```

*Listing 2.3 Sample returned data from `getSupportedConstraints` on a laptop computer*

The above data can be used by the web developer to check the capabilities of a device and adapt the `getUserMedia` call accordingly [59].

## WebRTC Peer Connection API

WebRTC uses the `RTCPeerConnection` interface to represent a connection between two peers and to handle efficient streaming of data between them [60]. An `RTCPeerConnection` object must be accompanied by configuration information which include an ICE agent, signaling state, ICE gathering state and ICE connection state. When the object is created the browser associates an ICE agent with the `RTCPeerConnection` object [61].



*Figure 2.4 RTCPeerConnection structure [62]*

The PeerConnection interface uses the ICE protocol together with the STUN and TURN servers to let UDP-based media streams to traverse NAT boxes and firewalls. ICE allows the browsers to discover enough information about the topology of the network where they are deployed to find the best exploitable communication path. Using ICE also provides a security measure, as it prevents untrusted web pages and applications from sending data to hosts that are not expecting to receive them [63]. An example of how to create an RTCPeerConnection object is shown in the following listing:

```
var config = {iceServers:
  [{url:'stun:stun.services.mozilla.com'}]};
pc = new RTCPeerConnection(config);
```

*Listing 2.4 Creating an RTCPeerConnection object*

In the above example, a new RTCPeerConnection object is created using the configuration described in the variable “config”. The variable declares one STUN server to be used by RTCPeerConnection. Once the RTCPeerConnection object is created by the browser it awaits for calls to methods createOffer, setLocalDescription, createAnswer, setRemoteDescription and addIceCandidate. These methods are summarized in the following table:

createOffer	Creates a request to find a remote peer with a specific configuration.
setLocalDescription	Changes the local description associated with the connection. The description defines the properties of the connection like its codec. The method takes three parameters, an RTCSessionDescription object to set, and two callbacks, one called if the change of description succeeds, another called if it fails.
setRemoteDescription	Changes the remote description associated with the connection. The description defines the properties of the connection like its codec. The method takes three parameters, an RTCSessionDescription object to set, and two callbacks, one called if the change of description succeeds, another called if it fails.

createAnswer	Creates an answer to the offer received by the remote peer, in a two-part offer/answer negotiation of a connection. The two first parameters are respectively success and error callbacks, the optional third one represent options for the answer to be created.
addIceCandidate	Provides a remote candidate to the ICE Agent. In addition to being added to the remote description, connectivity checks will be sent to the new candidates as long as the "IceTransports" constraint is not set to "none". This call will result in a change to the connection state of the ICE Agent, and may result in a change to media state if it results in different connectivity being established.

Table 2.2 RTCPeerConnection main methods [60]

For example to set the local description associated with the connection the peer connection will call its createOffer method which in turn will call a callback function which then calls the setLocalDescription method. Since a peer-to-peer connection has not yet been established at this point, the application also needs to notify the other peer through the signaling channel of its local description information.

```
pc.createOffer(function() {
    pc.setLocalDescription(sessionDescription);
    sc_send(sessionDescription);
}, onSignalingError, sdpConstraints);
```

Listing 2.5 Setting the local description using the createOffer method

The application must then wait for the remote description from the server and a list of ICE candidates as seen in the following listing:

```
socket.on('message', function (message){
if (message.type === 'answer') {
    pc.setRemoteDescription(new
RTCSessionDescription(message));
} else if (message.type === 'candidate') {
    pc.addIceCandidate(candidate);
}});
```

Listing 2.6 Setting the remote description and adding ICE Candidates

Another very important feature of the `RTCPeerConnection` API is its ability to support multiple streams per connection. For example two users can exchange the streams of their web cameras and also screen casting streams through the same media connection. Currently there are two different approaches for signaling multiple streams. Firefox since version 38 utilizes the “Unified Plan” [64] Internet draft while Chrome utilized an older Google proposal called “Plan B” [65]. As a result applications utilizing multiple streams are not interoperable between Chrome and Firefox [66]. Chrome development team has stated that they reevaluate this issue in the first quarter of 2016 [67].

The main goal of the “Unified Plan for Using SDP with Large Numbers of Media Flows” Internet draft is among others:

- ◆ To support for a large number of arbitrary sources
- ◆ To achieve glareless addition and removal of sources
- ◆ To avoid excessive use of port allocation

On Firefox, multiple streams can be implemented through the use of the `onnegotiationneeded` callback function of the `RTCPeerConnection` object. Every time a stream or track is added to an established `RTCPeerConnection` using the `addstream` method and `onaddstream` callback, it simply needs to be signaled to the other side of the connection using the `onnegotiationneeded` callback function as shown in the following listing:

```
if(pc) {
  pc.onnegotiationneeded = function (event) {
    pc.createOffer(
      setLocalDescription,
      onSignalingError,
      sdpConstraints);
  };
}
```

*Listing 2.7 Defining a callback function for when a new `RTCPeerConnection` negotiation is needed*

## WebRTC Data Channel

The WebRTC data channel API is designed to provide a transport service allowing web browsers to exchange generic data in a bidirectional peer-to-peer mode [63]. The WebRTC data channel is implemented by the `RTCDataChannel` interface which represents a bidirectional data channel between two peers of a connection [68]. `RTCDataChannel` can be configured to operate in different reliability modes. A reliable channel (the default `RTCDataChannel` connection) ensures that the data is delivered at the other peer through retransmissions [61].

The WebRTC data channel models the behavior of `WebSocket` [61] and is in fact a superset of the `WebSocket` API. Their main difference is that `WebSocket` runs on top of TCP whereas the WebRTC data channel is layered on top of three different protocols [62]:

- ◆ **UDP** which provides peer-to-peer connectivity.
- ◆ **DTLS** which provides encryption of transferred data.
- ◆ **SCTP** which provides multiplexing, flow and congestion control, and other features.

A comparison between the WebRTC Data Channel and `WebSocket` is summarized in the following table [62]:

	<b>WebSocket</b>	<b>DataChannel</b>
Encryption	configurable	always
Reliability	reliable	<b>configurable</b>
Delivery	ordered	<b>configurable</b>
Multiplexed	no (extension)	yes
Transmission	message-oriented	message-oriented
Binary transfers	yes	yes
UTF-8 transfers	yes	yes
Compression	no (extension)	no

*Table 2.3 Comparison between `WebSocket` and the WebRTC Data Channel*

Reliability and ordering can be set when creating the data channel by calling the `createDataChannel` method of the `RTCPeerConnection` object. These properties are members of an `RTCDataChannelInit` collection and are summarized in the following table:



maxPacketLifeTime	Limits the time during which the channel will transmit or retransmit data if not acknowledged. This value may be clamped if it exceeds the maximum value supported by the user agent.
maxRetransmits	Limits the number of times a channel will retransmit data if not successfully delivered. This value may be clamped if it exceeds the maximum value supported by the user agent.
negotiated	The default value of false tells the user agent to announce the channel in-band and instruct the other peer to dispatch a corresponding RTCDataChannel object. If set to true, it is up to the application to negotiate the channel and create an RTCDataChannel object with the same id at the other peer.
ordered	If set to false, data is allowed to be delivered out of order. The default value of true, guarantees that data will be delivered in order.

*Table 2.4 List of the most important members of the RTCDataChannelInit collection*

As it can be seen, there is no member in the above table that sets the reliability of the channel. An unreliable channel can be created either by limit the number of retransmissions (using member maxRetransmits) or by setting a time during which transmissions (including retransmissions) are allowed (using member maxPacketLifeTime). According to the specification these two properties cannot be used simultaneously and an attempt to do so will result in an error. Not setting any of these properties results in a reliable channel. This can be seen in the following 2 examples where we show how a reliable and an unreliable data channel can be created:

```
var con_init= {maxRetransmits: 0, ordered: false};
var data_channel = pc.createDataChannel("testChannel",
con_init);
```

*Listing 2.8 Creating an unreliable data channel*

```
var data_channel = pc.createDataChannel("testChannel", {});
```

*Listing 2.9 Creating a reliable data channel*

In the first example a data channel with unordered data transmission and no retransmits if a packet is not successfully transmitted while the second example shows the creation of a default reliable data channel.

## WebRTC Screen Casting

Screen capturing is an extension to the WebRTC `getUserMedia` API which allows the acquisition of a user's display, or part of it, in the form of a HTML5 video stream [69].

Screen capturing is achieved simply by calling the `getUserMedia` method using specific constraints. These constraints are specified by a `MediaStreamConstraints` object which has two members: `video` and `audio`, which describe the media type requested. An example constraints object is shown in the following listing.

```
constraints = {
  video: {
    mozMediaSource: "application",
    mediaSource: "application"
  }
};
```

*Listing 2.10 MediaStreamConstraints object for Application capturing in Firefox*

In the case of screen capturing the `audio` member is null because support for capturing audio from a native application is not yet implemented. The `video` member specifies the requested type of screen capturing. These types can be depending on the browser, one of the following:

- ◆ A **monitor** surface: Represents the full screen area for one of the connected screens (e.g. the user is asked to select a monitor in a system with multiple monitors connected). It can also represent a combination of all the connected screens.
- ◆ A **window** surface: Captures a single window. Child windows (modal or not) will not be captured
- ◆ An **application** surface: This surface represents all the windows that are available to a single application. All child windows modal or not will be captured. An application capture results in a surface area with dimensions

equal to the user's screen resolution but with all areas that don't belong in the captured application obscured.

- ◆ A **browser** surface: This represents a single document. It is usually a browser tab but it is not strictly limited to HTML. Theoretically it could be a single document from any application, although this feature has not yet implemented by any browser.

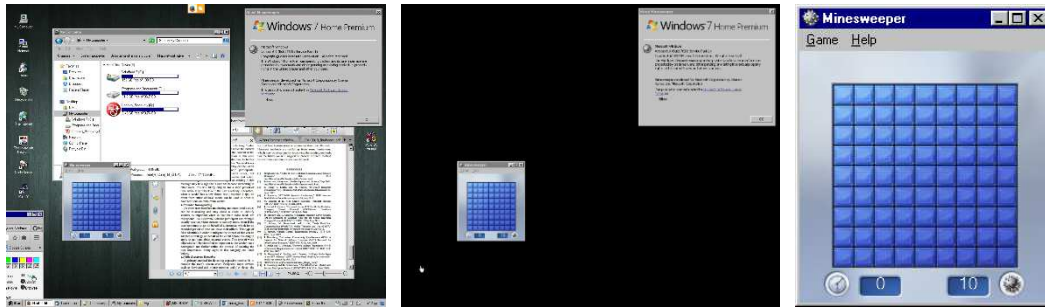


Figure 2.5 Screen capturing modes (from left to right: Monitor, Application, and Window) Notice how the "About" dialog box is visible in the 2<sup>nd</sup> screenshot (Application capturing) because it belongs to the Minesweeper Application

As of February 2016 the screen capturing API is only supported on Chrome and Firefox. On Firefox in order for a web application to be allowed to access the screen capturing API it must a) be behind an SSL server (https) and b) has its domain whitelisted in the `media.getusermedia.screensharing.allowed_domains` variable of the Firefox application settings (about:config). A simple extension can be developed to automatically whitelisting a domain but developers of commercial WebRTC screen sharing applications can also request their domain to be whitelisted in the next release of Firefox. Firefox currently doesn't support capturing of browser documents (tabs).

On Chrome screensharing is implemented as an API for Chrome extensions which must be installed through the Chrome Web Store. The API is much more difficult to use and documentation for it is currently very sparse. Although this defies the philosophy of plugin-less web advocated by WebRTC, Google decided to enforce this policy of security reasons. Because extensions for Chrome can only be delivered through the Chrome Web Store, Google is allowed to maintain some form of governance, along with the ability to throw away extensions that are considered malware [70].

## Screen Casting Security Issues

Screen sharing in the browser has significant security implications, the most obvious being users sharing content that they did not wish to share, or users not realizing that they are actively sharing their screen or portion of it. Also display of information that is under the control of the browser (e.g. a browser tab that the web application has access to) can allow the web application to access information that would otherwise be inaccessible to it directly [71] and thus render the “same-origin policy” inefficient. To summarize, the main security concerns of W3C are a) the capture of an area that is not intended to be exposed and b) the capturing of an area without the authorization of the user.

To secure against capturing of surfaces areas that the user has not provided capturing authorization the browser can obscure some of the captured areas as shown in the following screenshot: Here a user has chosen to capture the Windows Notepad application. The browser is obscuring the rest of the desktop only displaying the application and its child windows.

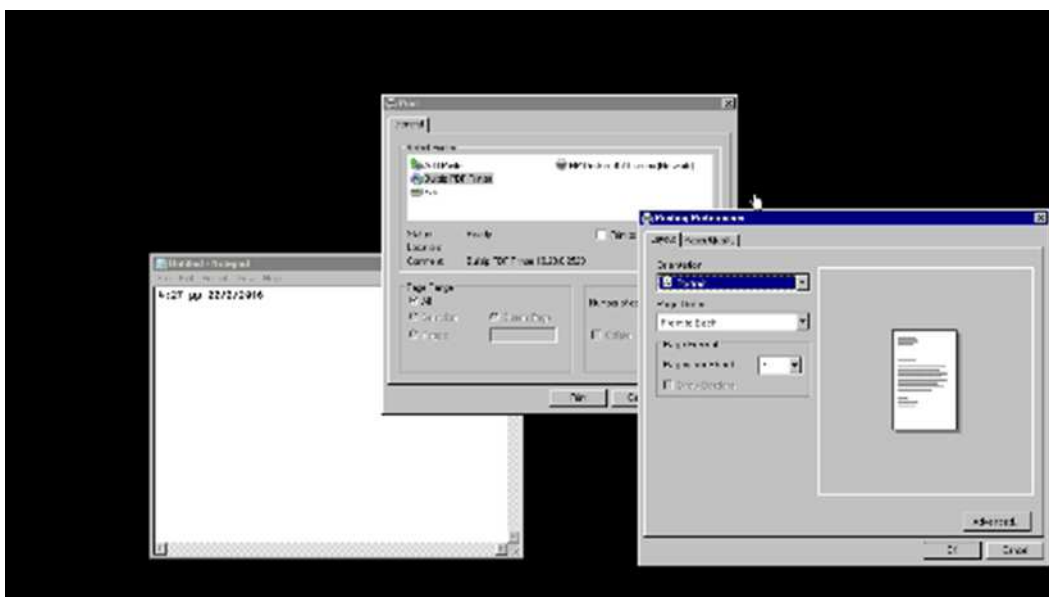
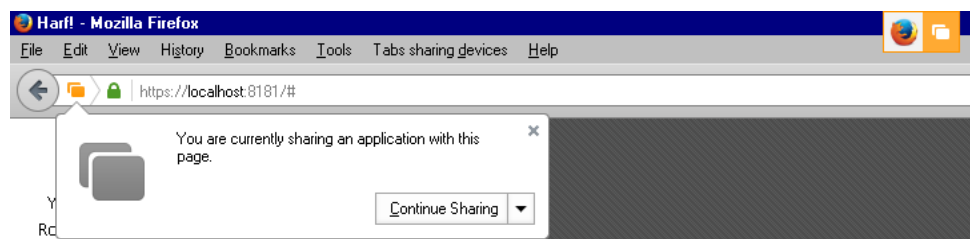


Figure 2.6 Obscuring information in application capturing mode (black area covers everything that doesn't belong to the application)

To secure against initiating a screen sharing session without the explicit authorization of the user the draft specifies two different forms of user interaction. In the first case the application (e.g. browser) that is requesting to use the screen capturing feature has no control over what is to be rendered on that surface. In this

case active user consent (e.g. by selecting an application from a drop down menu) is all that is required. The user must remain notified that a screen sharing session is active (e.g. by an always-on-top notification such as the one shown in the next figure). The user must also be able to stop any active capture at any time. Firefox for example displays an always-on-top orange square with the Firefox icon on the middle top of the screen, and an icon in the address bar. When the user clicks on either of these areas, a popup notification appears indicating that a device is used or that part of the screen is captured, from which the user can stop the capturing session, as seen in the following figure.



*Figure 2.7 Notifying the user that screen sharing is active in Firefox*

In cases where the application has control over the area that is about to be shared, the W3C draft strongly advises for a form of “elevated permission” to be required from the user. This “elevated permission” should, among others things, notify the user of the risks associated with enabling screen capturing and certify that the user has trust in the application.

The vulnerabilities arising from the lack of enforcement of the “same-origin policy” by WebRTC screen capturing are discussed extensively in [72] by Tian et al.

The same-origin policy is a critical web security mechanism for isolating potentially malicious documents. It restricts a document or script loaded from one origin from interacting with a resource from another origin [73]. Tian et al. conclude that this assumption is directly broken when using the screen sharing feature of WebRTC and distinguish between malicious users and malicious WebRTC applications. A malicious user is one who tries to collect sensitive information by tricking a benign user to click on malicious links during a screen sharing session. A malicious WebRTC application is one which can access the cross-origin content displayed inside a user’s browser. Utilizing this technique an attacker could perform attacks on integrity (CSRF) and attacks on confidentiality (access to personal

information, browsing history etc.). For example a malicious user or malicious application could trick the user into clicking a malicious link to an HTML file containing the following code:

```
<script>
document.location="view-source:https://www.facebook.com/";
</script>
```

Listing 2.11 Malicious HTML file for a possible CSRF attack through screen sharing

The above code displays the source code of Facebook.com (fig. 3). The attacker now has obtained a screenshot of the source code of the website which could contain critical information such as validation tokens etc.

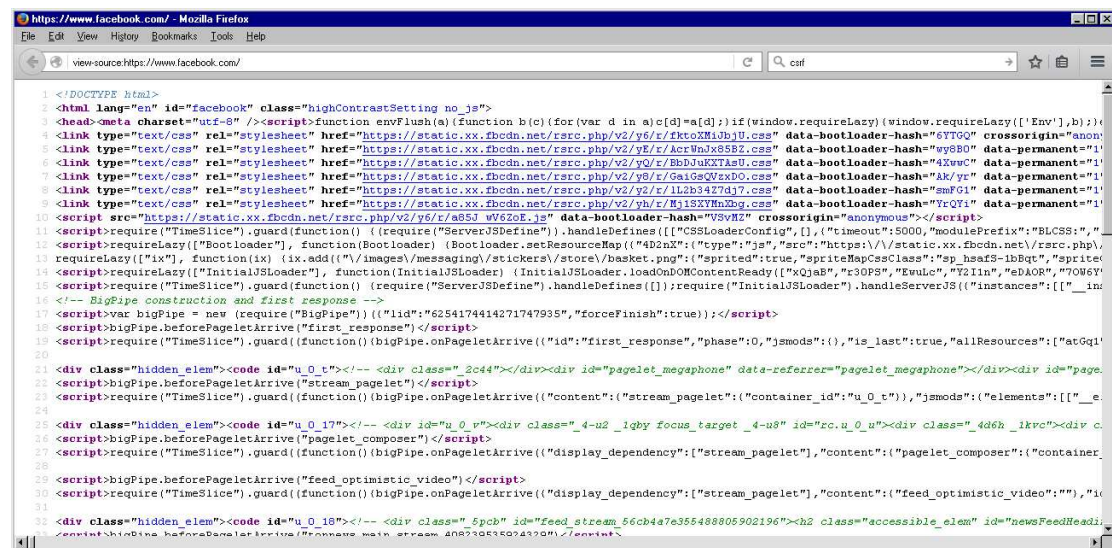


Figure 2.8 Source code of a website which could contain security-critical information such as security tokens which could be used for a CSRF attack.

## WebRTC Interoperability

Until the WebRTC standard is more finalized, the two supporting browsers (Firefox and Chrome) could be using different prefixes for their WebRTC interface.

Some differences in the WebRTC API between Firefox and Chrome are summarized in the following table (as of September 2015):

W3C Standard	Chrome	Firefox
getUserMedia	webkitGetUserMedia	mozGetUserMedia
RTCPeerConnection	webkitRTCPeerConnection	mozRTCPeerConnection
RTCSessionDescription	RTCSessionDescription	mozRTCSessionDescription
RTCIceCandidate	RTCIceCandidate	mozRTCIceCandidate

Table 2.5 Example API differences across Chrome and Firefox

To remedy this and to help ease cross-browser development the WebRTC group has developed a “polyfill” shim library (a small library that transparently intercepts API calls and changes the arguments passed, handles the operation itself, or redirects the operation elsewhere) called “Adapter.js”. This library helps insulate apps from cross-browser API differences by letting developers write code using W3C standard names [34].

The following code shows an example of how Adapter.js works. In this example it is shown how Adapter.js detects the user’s browser and redefines the object `RTCPeerConnection` as “`mozRTCPeerConnection`”:

```
if (navigator.mozGetUserMedia) {  
  console.log("This appears to be Firefox");  
  webRTCdetectedBrowser = "firefox";  
  // The RTCPeerConnection object.  
  RTCPeerConnection = mozRTCPeerConnection;  
}
```

*Listing 2.12 Detecting the browser and intercepting WebRTC API accordingly*

Adapter.js is being developed at [github.com/webrtc/adapter](https://github.com/webrtc/adapter)

It must be noted here that since December 2015 the method `getUserMedia` belongs to the `MediaDevices` interface instead of the `Navigator` interface that is shown on the previous listing. The `Navigator` interface represents the state and the identity of the user agent. It allows scripts to query it and to register themselves to carry on some activities. On the other side, the `MediaDevices` interface in which the `getUserMedia` method now belongs is a specialized interface that provides access to connected media input devices like cameras and microphones, as well as screen sharing. This interface currently has 2 main methods: `getUserMedia()` which has been discussed in extend in previous section and an `enumerateDevices()` method for obtaining arrays of information about the media input and output devices available on the system [74] [75] [76]. Adapter.js mitigates this in old browser versions as seen in the following listing:

```
if (!navigator.mediaDevices) {  
  navigator.mediaDevices = {getUserMedia: requestUserMedia,  
                             addEventListener: function() { },
```

```

    removeEventListener: function() { }
  };
}

```

Listing 2.13 Mitigating the lack of `MediaDevices` interface in `Adapter.js`

### 2.2.3 WebRTC Signaling

Although WebRTC aspires to enable Peer-to-Peer communication between browsers without relaying data through a server, a use of a server is still required for two reasons: The first reason is the obvious one, the server is needed to “serve” the actual JavaScript application that utilizes WebRTC. The second reason is less obvious. A server is required in order to initialize sessions between the clients that need to communicate. This process is known as “Negotiation” and is implemented via certain signaling exchange. The latter is responsible for the exchange of the initial (meta) data of session descriptions (using SDP) which contain details on the form and nature of the data which will be transmitted [77]. This information can include [78]:

- ◆ Network data, such as IP addresses and ports.
- ◆ Media metadata such as codecs and codec settings, bandwidth and media types.
- ◆ Error messages.
- ◆ User and room information.

The following schematic shows the signaling architecture of WebRTC:

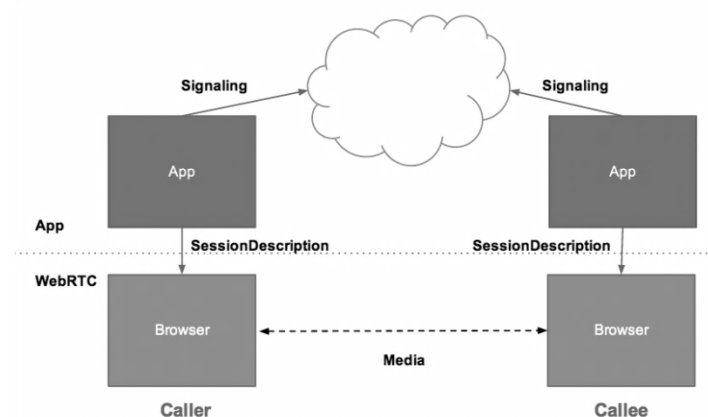


Figure 2.9 WebRTC Signaling Architecture

The WebRTC signaling process is based on a new standard called JSEP (JavaScript Session Establishment Protocol). JSEP is a collection of interfaces used to



identify negotiation of local and remote addresses by exchanging “offers” and “answers” between peers using SDP.

The signaling itself in WebRTC is completely abstract and not defined in any specification. The reason behind this is that different applications may decide to use different signaling protocols such as SIP for e.g. VoIP applications, XMPP for e.g. chat applications, HTTPS (WebSocket or socket.io) for e.g. plain web applications such as the one we present in this thesis or even something special for a novel use case [79]. In other words, standardizing on the wrong signaling protocol could easily limit the future potential of WebRTC.

More details about SDP and the signaling process will be discussed in section 2.2.4 concerning WebRTC Network Protocols. In the next section we discuss the signaling server implementation we chose in this thesis which utilizes WebSocket (using socket.io).

## **Node.js/Socket.io for WebRTC Signaling**

One of the most popular implementations for WebRTC signaling is using Socket.io. Socket.io is a JavaScript library for real time web applications that supports bi-directional event-based communication. It has two parts: a client-side JavaScript library that runs in the browser and a server-side library for node.js.

Node.js is a JavaScript framework that simplifies the writing of event-driven server-side applications using a build-in HTTP server implementation. It is based on a single-threaded event loop management process making use of non-blocking I/O. With Node.js, it is really easy for the programmer to implement a high-performance HTTP server with customized behavior with just a few lines of JavaScript code [63].

Socket.io although it is utilizing the WebSocket protocol, it is more than a simple WebSocket wrapper as it offers many other features such as broadcasting to multiple sockets and support for “Rooms” which are essential for most WebRTC applications. An example server and client is shown in the following listings:

```
var io = require('socket.io').listen(80);
io.sockets.on('connection', function (socket) {
  socket.emit('event1', 'hello from server!');
  socket.on('event2', function (data) {console.log(data);});
});
```

*Listing 2.14 Socket.io on the server side*

```
var socket = io.connect('http://localhost');
socket.on('event1', function (data) {
  console.log(data);
  socket.emit('event2', 'hello from client!');
});
```

*Listing 2.15 Socket.io on the client side*

The above listings show the simplicity and elegance of socket.io and how it simplifies event-driven web development. On the server side socket.io listens to port 80 and upon connection emits a ‘hello from server’ event called ‘event1’ which fires the event1 event on the client. The event1 event prints the data that accompanies the event and then sends an even2 on the server with the string “Hello from client”.

## 2.2.4 WebRTC Network Protocols

WebRTC relies on a number of protocols to be able to communicate with other clients. Before a peer-to-peer connection is established, all peers must exchange session descriptions which contain information about the peers (e.g. IP addresses) and the type of information they wish to exchange (e.g. the type video codecs to be used). This information is exchanged through the signaling server using SDP. Once session information is acquired by all peers, the actual peer-to-peer connection must be established. Because naturally some of the peers can be behind a NAT, WebRTC must also implement some form of NAT traversal. This is possible with the use of STUN and TURN which are part of the ICE protocol. In this section we take a close look at these protocols.

### SDP

SDP (Session Description Protocol) is a format for describing streaming media initialization parameters. It is published as an IETF proposed standard in RFC 4566 in 2006. SDP provides a standard representation for such information, irrespective of how that information is transported. It is intended to be general purpose protocol so that it can be used in a wide range of network environments and applications [80]. A session is described by a series of fields, one per line, each line being in the form of *character=value*, where *character* is a single-case significant character and *value* is structured text whose format depends on the attribute type [81]. For example the

character m signifies media name and transport address and the letter o is the originator and session identifier. In the following example we show some lines of an SDP message generated by Firefox for a WebRTC session:

```
v=0
o=mozilla...THIS_IS_SDPARTA-44.0 8584189520789373760 0 IN IP4 0.0.0.0.
a=ice-options:trickle
a=msid-semantic:WMS *
m=audio 9 UDP/TLS/RTP/SAVPF 109.
a=rtpmap:109 opus/48000/2
a=rtpmap:9 G722/8000/1
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000.
a=rtpmap:120 VP8/90000
a=rtpmap:126 H264/90000
a=rtpmap:97 H264/90000
```

*Listing 2.16 Excerpt of SDP message*

In the above example, the lines beginning with character “a” (media attribute lines — overriding the Session attribute lines) is the list of available audio and video codecs of the browser from whom this message originated (in this case VP8 and H264 for video and Opus, G.722, PCMU and PCMA). The originating browser is Firefox 44.0 as seen from the value of character “o” (originator) being equal to “mozilla...THIS\_IS\_SDPARTA-44.0” (a humorous reference to the quote “This is Sparta!” from the movie ‘300’).

## STUN

STUN is one of the protocols used by ICE that serves as a tool for other protocols in dealing with NAT traversal, standardized in 2008 as RFC 5389 [82]. Most times, a client behind a NAT is unaware of its public IP address and port. To resolve this, the client sends a message to a STUN server (which is located behind the NAT) on the public web. The STUN server then sends a reply containing the client’s public IP and port as seen from its side.

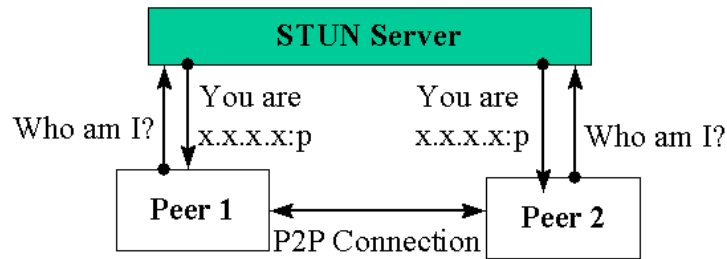


Figure 2.10 Connection using a STUN Server

Because STUN is a very light-weight and simple protocol, STUN server with low specifications can handle a large number of requests [83]. It is measured that 86% of WebRTC connections are successfully completed using STUN.

## TURN

On some occasions, WebRTC can fail to establish a connection using STUN. This usually happens when one of the clients is behind a symmetric NAT because with a symmetric NAT a client can find out its public IP address but not its public port. To be reachable, a device behind a symmetric NAT needs to initiate and maintain a connection using relay [84]. WebRTC uses the TURN protocol (Traversal Using Relays around NAT) to establish a relayed connection. TURN is meant to bypass the symmetric NAT restriction by opening a connection with a TURN server and relaying all information through that server.

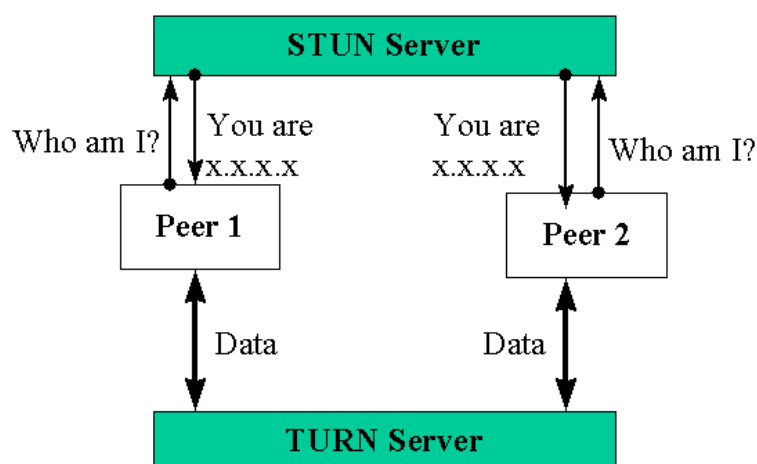


Figure 2.11 Using a TURN Server to relay data

It is obvious that this fallback solution comes with some overhead so it is only used if there are no other alternatives [85].

## ICE

The STUN and TURN protocols are all part of the ICE (Interactive Connectivity Establishment) framework that is used by WebRTC. ICE is used to allow browsers to connect with other browsers (peers).

An example of using ICE to tell the `RTCPeerConnection` object which STUN and TURN servers to use is shown in the following listing:

```
var stun_server = {'url': 'stun:stun.services.mozilla.com',};
var turn_server = {
  url: 'turn:nikos@numb.viagenie.ca', credential: 'pwd12345'
};
var iceServers = {iceServers: [stun_server, turn_server]};
var pc = new RTCPeerConnection(iceServers);
```

*Listing 2.17 Creating an `RTCPeerConnection` object and telling it which STUN and TURN servers to use*

ICE is described in RFC 5245, “Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols” [86]. The basic idea behind ICE is that each peer has a list of candidate IP addresses and ports it could use to communicate with the other peer. These are called “candidate addresses”. Candidates include the host’s private local IP address and also IP addresses collected from STUN and TURN servers. ICE then distributes the candidate addresses to all the other peers using the signaling server. Finally each peer attempts to connect to the other using the candidates in its list. Each of the candidates is assigned a priority value. Lowest priority is given to the relayed candidates, highest priority to local candidates.

The following two tables shows an example of ICE candidates gathered for a sample WebRTC session:

Local Candidate	Remote Candidate	ICE State	Priority	Nominated	Selected
10.18.0.84:56308/udp(host)	64.95.96.8:33526/udp(host)	failed	9115005270299247000		
147.95.122.168:60396/udp(peerreflexive)	64.95.96.8:33526/udp(host)	succeeded	7962083765692400000	true	true
64.95.96.21:40146/udp(relayed-udp)	64.95.96.8:33526/udp(host)	failed	396070476570427400		
64.95.96.21:14320/udp(relayed-tcp)	64.95.96.8:33526/udp(host)	failed	35782506380787710		
147.95.122.168:53888/udp(serverreflexive)					
	64.95.96.8:33526/udp(host)				

Figure 2.12 Sample list of local and remote ICE candidates gathered by Mozilla Firefox (the public addresses of the peers are paired)

Local Candidate	Remote Candidate	ICE State	Priority	Nominated	Selected
10.0.29.218:54777/udp(host)	10.0.29.55:53666/udp(peerreflexive)	succeeded	7962083765675491000	true	true
147.95.122.204:58119/udp(serverreflexive)					

Figure 2.13 List of local and remote ICE candidates gatered by Mozilla Firefox (local IP of the peers is selected)

One security implication of using ICE is that it exposes IP information by allowing requests to STUN servers that return the local and public IP addresses of the user using JavaScript. Additionally, because these STUN requests are made outside of the normal XMLHttpRequest procedure, they are not visible in the developer console or able to be blocked by plugins such as AdBlockPlus or Ghostery. This makes these types of requests available for online tracking if an advertiser sets up a STUN server with a wildcard domain [87]. A demo of this vulnerability can be seen in the following screenshot:

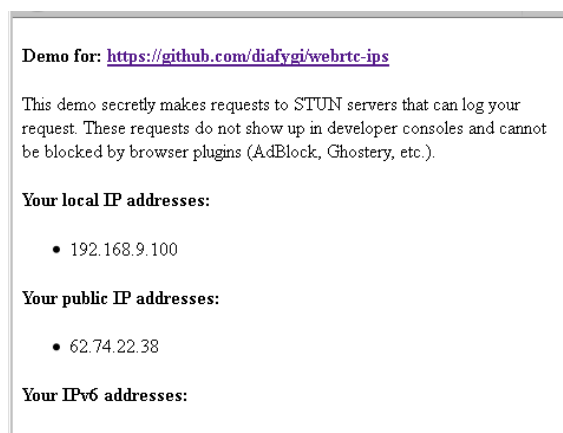


Figure 2.14 Using WebRTC to reveal user's IP information

The whole process of utilizing the protocols we discussed in this section establishing a connection between two peers in WebRTC is shown in the following diagram:

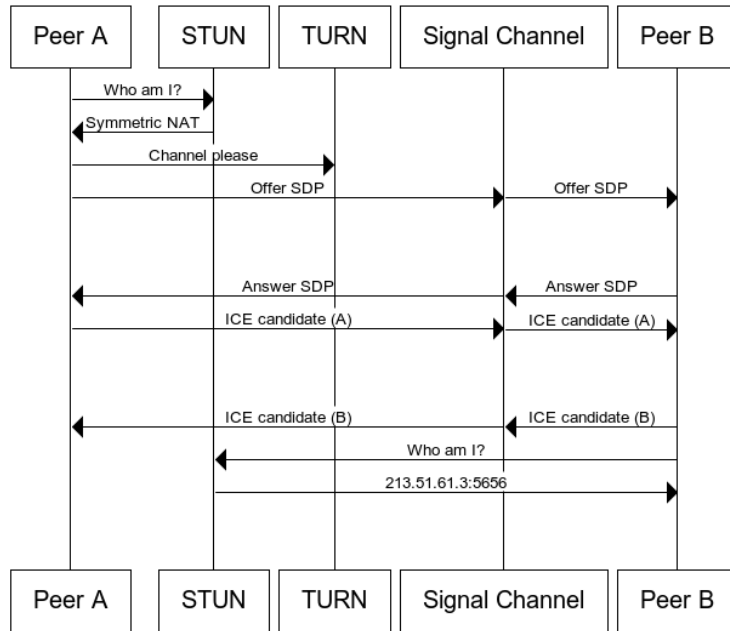


Figure 2.15 WebRTC signaling complete process [88]

## 2.2.5 WebRTC Codecs

The codecs that are supported by WebRTC are defined by two IETF drafts. Audio codecs are described in “WebRTC Audio Codec and Processing Requirements” [37] and video codecs in “WebRTC Video Processing and Codec Requirements” [36]. According to these drafts WebRTC browsers must (absolute requirement) implement the VP8 video codec as described in RFC6386 and also H.264 Constrained Baseline as described in H264, and must also implement the Opus audio codec described in RFC6716 and the G.711 PCMA and PCMU audio codec described in RFC3551. WebRTC also supports the iSAC and iLBC audio codecs.

VP8 is a video compression format which is owned by Google and is the video codec of the WebM file format. The benefits of VP8 are its low bandwidth requirements and its broad spectrum of supported hardware from desktop computers to embedded devices. VP8 is designed to make the optimal use of computation power in modern hardware while maintaining fast decoding speeds [89]. Although VP8 is influenced by H.264/AVC it includes several technological innovations.

Although VP8 is the video codec of choice of WebRTC, it was decided by IETF in November 2014 that support of H.264 will also be mandatory. H.264 or MPEG-4 Part 10, Advanced Video Coding (MPEG-4 AVC) is a video coding format that is one of the most commonly used formats for the recording, compression, and

distribution of video content. This popularity of H.264 is one of the reasons that led to the decision of making it a mandatory WebRTC codec [90]. It is currently only supported by Firefox [31]. Compared to VP8, H.264 offers better video quality at the same bit rate especially in higher motion videos [91].

Opus is the primary audio codec of WebRTC. It is an open source and royalty free audio codec intended for storage and streaming, standardized by IETF in RFC6716. It supports bitrates from 6 to 510 kbps, frame sizes from 2.5 to 60 ms, sampling rates from 8 to 48 KHz, dynamically adjustable bitrate [92], has an audio bandwidth ranging from narrowband (0.3 to 3.4KHz) to full band and was developed specifically for packet switching networks.

The G.711 is a very simple ITU-T recommendation dating from 1972 which was designed to carry audio at a fixed bitrate of 64kbps and is used for many years in packet switched networks. It is a narrowband codec (0.3 to 3.4 KHz) which means can only be used for voice applications. G.711 does not implement any compression, hence it has zero compression latency [93]. G.711 is included in the list of mandatory WebRTC codecs mainly for legacy reasons [94].

Finally, the iSAC (Internet Speech Audio Codec) and iLBC (Internet Low Bitrate Codec) are voice codecs which although are not mandatory are supported today by some platforms. iSAC is a bandwidth-adaptive, wideband and super-wideband voice codec suitable for streaming audio and VoIP applications with a sampling frequency of up to 16KHz and adaptive bitrate, while iLBC is narrowband with a fixed bitrate. Both are royalty-free.

## **2.3 HTML5**

The original HTML was proposed and prototyped in the early 1990s by Tim Burners Lee. Ever since it has been in continuous development. Some features were introduced in specifications while others were introduced in software releases. HTML4 became a W3C Recommendation in 1997.

The current proposed draft is HTML5 which brings to the Web, video and audio without the need of plugins, programmatic access to a bitmap canvas, useful for rendering graphs, game graphics, or other visual images on the fly [95] (procedural animation/drawing), native support for scalable vector graphics (SVG) and math (MathML), features to enable accessibility of rich applications and much more [96].



In essence HTML5 helped the Web evolve from a presentation layer powered by a simple markup language to a full-scale application environment.

The HTML5 draft reflects an effort that started in 2004, to study contemporary HTML implementations and deployed content [97]. The draft:

1. Defines a single language called HTML5 which can be written in HTML syntax and in XML syntax.
2. Defines detailed processing models to foster interoperable implementations.
3. Improves markup for documents.
4. Introduces markup and APIs for emerging idioms, such as Web applications and these include for example the media elements (audio, video) and the canvas element.

HTML5 was standardized in 28 October 2014, and the current working draft 5.1 is scheduled to be released by the end of 2016.

### **2.3.1 HTML5 APIs**

The development and introduction of a plethora of APIs in HTML5 rose from the need to provide users with native-like experiences within the browser [98]. For example, there are now many APIs that can access mobile devices on the hardware level and report battery status, vibrate the device and even measure the ambient light of the environment on devices equipped with a light detector. Things which in the past would require time-consuming development of browser plugins are now possible with a few lines of JavaScript code.

The core API of this thesis is the WebRTC API which was described in the previous section. Other important HTML5 APIs used in this thesis include the File API, the Stream Capture from DOM Elements API and the Media Recording API which we will describe in the following paragraphs.

### **File API and the Blob Interface**

One area in which the Web lacked for some time is file I/O. Interacting with local data is the core of most desktop software, but for web application this was not possible until the introduction of the HTML5 File API [99].

The File API makes it possible for browsers to access data from the underlying operating system and manipulate that data before sending it to either a web server or a peer browser. The specification of the File API defines basic representations for files, list of files, errors caused by access to files and programmatic ways to read files. The File API also includes the “Blob” interface which represents immutable raw data [100].

As stated the File API is capable of reading and writing data to the user’s hard disk. For this reason a number of security considerations are taken into account in the draft. Those include for example:

1. **Storing malicious executables on the user’s system.** The API tries to prevent this by restricting file creation and file renaming to non-executable file formats. The API also makes sure that the execute bit is not set on any file it creates or modifies [99].
2. **Leakage or deletion of user data.** The specification assumes that the primary user interaction is with the HTML input element and that all files that are being read by the API have first been selected by the user [100]. The API also prevents access to system sensitive files.

## **Stream Capture from DOM Elements API**

Capturing media streams from DOM elements is a W3C draft that was published in 19 February 2015 by the WebRTC and the Device API working groups with no revisions since. It is intended to become a W3C recommendation at some time in the future. The draft describes an extension to the HTML media and canvas elements that enables capturing the output of the element in the form of streaming media. The stream can in turn be broadcast through the WebRTC media channel, recorded or otherwise used by any other HTML5 APIs that handle media streams such as WebAudio [101].

Implementations of the stream capturing API are still highly experimental. In Firefox the `captureStream()` method is implemented for the `HTMLCanvasElement` object since release 43 (15 December 2015) [102]. The `HTMLMediaElement` on the other side has an undocumented method `mozCaptureStream()` that is implemented [103]. The method is prefixed with the `moz-` prefix and is problematic in its use most

notable problems being those documented in bugs 1178751 and 912907 of the Mozilla Bug database. We will further explore these problem in chapter 3.

In Chrome support for this API has not yet been released. An initial implementation is scheduled for Chrome 50 which is due to be released in 11 April 2016. As of February 2016 (Chrome Canary) audio was still not supported when capturing streams from video elements and there were other documented bugs such as canvases in background tabs do not update properly [104].

## Media Recording API

The capability to record media in HTML5 has been suggested by W3C in a working draft by the Device APIs working group and the WebRTC working group, initially published in 5 February 2013. The API allows very basic stream recording in the browser while also allowing for more complex use cases. The API provides the developer with a *MediaRecorder* object with *record()* and *stop()* methods and an *ondataavailable* event that is fired when the recording has stopped and recorded data is available in the form of a HTML5 blob. Functions are also available to query the platform's available set of encodings, and to select the desired ones if the author wishes [105].

The Media Recorder is still experimental in most browsers and as a result its use is still problematic. It is expected that the API implementations will be improved as new browser versions are released. The current status of the API (as of February 2016) is presented in the following table [106]:

Browser	Version	Comments
Chrome	47	Feature not enabled by default. Currently only video is supported.
Firefox	25	Initially only supported audio recording. Currently supports both video and audio but its use is problematic
Firefox Mobile	25	
Firefox OS	1.3	

### 2.3.2 jQuery & jQueryUI

jQuery is a JavaScript library that simplifies HTML document traversing, event handling, animating and Ajax interaction [107]. It exposes a “tree-query language” API which allows the developer to achieve three things: traverse the DOM and select an initial set of nodes of it, navigate to nodes relative to those nodes and more importantly manipulate these nodes easily and uniformly [108]. jQuery is popular not only because of its ease of use but also because it helps separate design from structure and also behavior from structure within a HTML document. This approach of development is known as *Unobtrusive JavaScript* [109] [110]. Unobtrusive JavaScript leads to clean and semantic HTML and more manageable code. An example can be seen in the following code:

```
| <a href="#" onclick="doSomething();" id="button1">
```

*Listing 2.18 HTML containing obtrusive JavaScript*

```
| <a href="#" id="button1">
```

*Listing 2.19 Same HTML code with JavaScript now removed*

```
| var button1=document.getElementById('button1');  
| button1.onclick = doSomething;
```

*Listing 2.20 Unobtrusive JavaScript located in a separate .js file*

```
| $("#button1").on("click", doSomething);
```

*Listing 2.21 Unobtrusive JavaScript located in a separate .js file using jQuery*

jQuery is omnipresent in the modern World Wide Web, used by about 70% of all websites (fig.2.16) with a market share of almost 96% among JavaScript libraries [111].

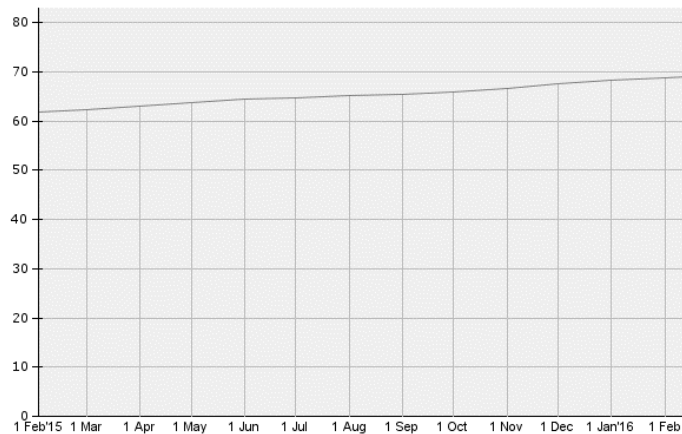


Figure 2.16 Usage of jQuery for websites, 15 Feb 2015 to 15 Feb 2016. Source: W3Techs.com

jQueryUI is another JavaScript library that is built on top of jQuery. It simplifies the development of user interface interactions that are required for modern web applications (especially single page web sites such as the one that was implemented in this thesis). It provides user interface interaction elements, animation effects, widgets and themes.

## 2.4 Data Chanel Compression

In order to save bandwidth, we propose a compression system on the WebRTC data channel based on the LZW compression algorithm. All data travelling through the data channel are compressed using LZW beforehand and immediately decompressed upon arrival from the other peer. This is shown in the following schematic which presents a data transfer of data from Peer1 to Peer2.

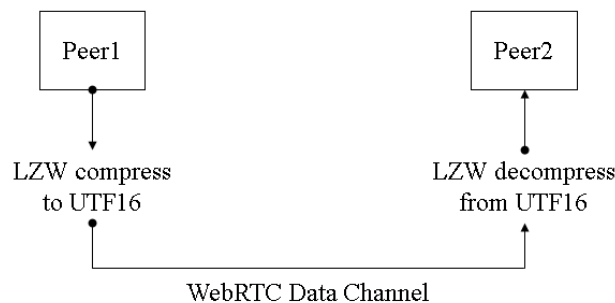


Figure 2.17 Data channel compression schematic

We evaluate the performance of this in implementation in comparison with a compressionless data channel in chapter 5. We present implementation details in chapter 4. In this section we simply present some details about the LZW algorithm.

Lempel–Ziv–Welch (LZW) is a universal lossless data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch. It was published by Welch in 1984 as an improved implementation of the LZ78 algorithm published by Lempel and Ziv in 1978. The algorithm is simple to implement, and has the potential for very high throughput in hardware implementations [112].

LZW is organized around a string table (dictionary). At each stage of the algorithm bytes are gathered into a sequence. This continues until the next character gathered forms a sequence which is not included in the string table.

The algorithm flowchart is presented in the following figure [113]:

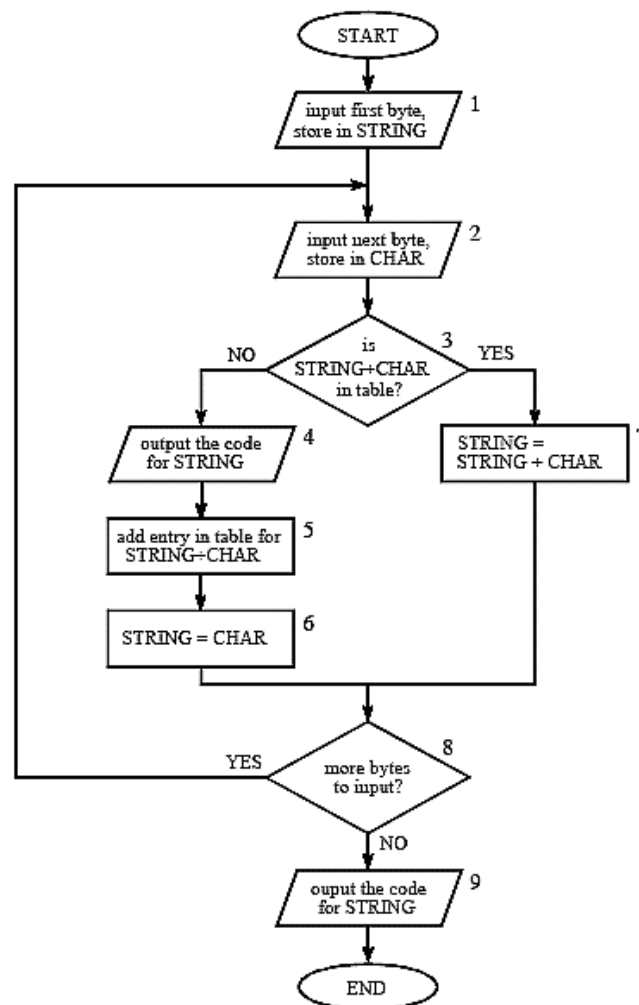


Figure 2.18 LZW Algorithm Flowchart

## Chapter 3. Communication Protocol

Before developing a system targeted on synchronous online collaboration, a protocol defining the messages that will be exchanged through the WebRTC data channel must be developed. This is necessary if the system is going to be expandable, interoperable and maintainable. What we propose is a “language” defining the actions taking place in such a collaboration environment and their parameters.

The first step towards defining this “language” targeted at synchronous online collaboration is to develop some form of abstraction layer sitting on top of the native WebRTC `RTCDataChannel` interface which was described in Chapter 2. There are two basic reasons for this: first, we need functions to uniformly handle messages exchanged between peers and second, because the use of the internal WebRTC functions to exchange data through the data channel is often a complicated task requiring many lines of code and customizations. In chapter 2.2.1 we discussed WebRTC API wrappers. In a similar fashion we propose wrapper functions encapsulating the data channel functionality into simple send and receive functions which simplify the development and maintenance of the system. For these reasons we wrote two functions: One for sending messages (called `sendDataAction`) and one for automatically handling incoming messages (called `handleMessage`). We describe these two functions in chapters 3.2 and 3.3 respectively.

Once this WebRTC abstraction layer is developed, the next step is to define a simple “language” that will describe actions in a synchronous collaboration environment (such as sketches or user chat messages) and their parameters. The language we propose consists of character strings and “stringified” JSON objects. We examine this proposed language in detail in chapter 3.3. Finally we give an example of how the system can be easily expanded by using this language, in chapter 3.4

The communication model described above is shown in figure 3.1. The foundation of the system is the native WebRTC data channel `RTCDataChannel.send()` function and `onmessage` property. Above this layer we have the abstraction layer consisting of the `sendDataAction()` and `handleMessage()` functions. Finally the top layer consists of the exchanged standardized messages which represent actions and function calls.

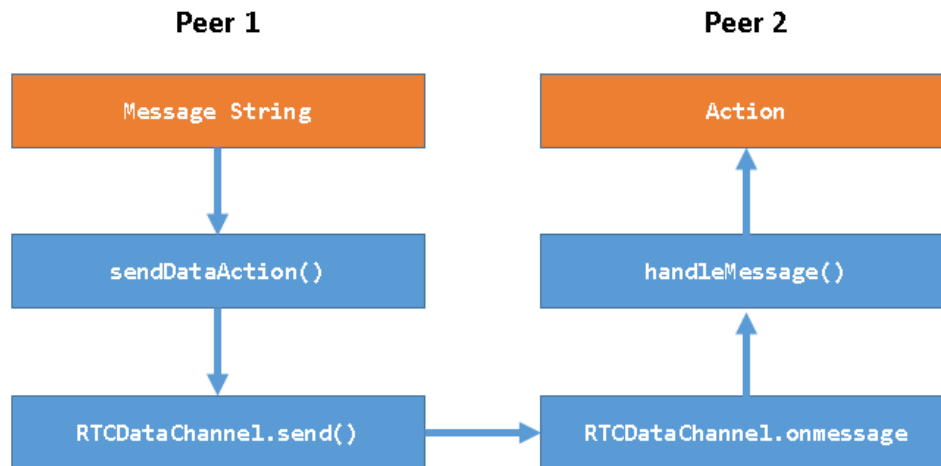


Figure 3.1 Communication Model

### 3.1 Use of the native WebRTC send/receive functions

The abstraction layer we will create, at its base will employ the native WebRTC functions used for sending and receiving data through the data channel. These methods were discussed in detail in chapter 2. To summarize them here, the `send()` method of the `RTCDataChannel` interface is responsible for sending data across the data channel to the remote peer at any time except during the initial process of creating the underlying transport channel. Data sent before connecting is buffered if possible (or an error occurs if it's not possible), and is also buffered if sent while the connection is closing or closed. Received data is handled by a call back function which is defined in the `RTCDataChannel` property “`onmessage`”. This property stores an event handler which specifies a function to be called when the message event is fired on the channel. This event is represented by the `MessageEvent` interface. This event is sent to the channel when a message is received from the other peer.

The basic use of the `RTCDataChannel.send()` method and the call back function “`onmessage`” used when messages are received is outlined in the following code:

```

var pc = new RTCPeerConnection();
var dc = pc.createDataChannel("BackChannel");

function sendMessage(msg) {
  // sample JSON object
  let obj = { "message": msg, "timestamp": new Date() }
}
  
```



```

    // Convert JSON to string and pass it to the data channel
    dc.send(JSON.stringify(obj));
}

dc.onmessage = handleMessage;

```

*Listing 3.1 Usage of the RTCDataChannel send() method and the onmessage callback.*

### 3.2 Sending data with sendDataAction

As we discussed in the beginning of this chapter, we need to develop an abstraction layer encapsulating the WebRTC methods for sending and receiving data through the data channel. For sending data we have developed a function called sendDataAction() for sending strings and a function called sendDataFile() for sending binary data.

All data and messages are sent using these two functions. Messages are in turn received and handled inside a callback function called handleMessage(string message, bool compression) which we will describe in chapter 3.3.

The sendDataAction() function has two arguments: The first is a string containing the data to be exchanged, while the compression argument is an optional boolean defining whether the incoming message is compressed using the LZW algorithm (true) or not compressed (default, false). In the same manner the first argument of the sendDataFile() function is a blob containing the binary data to be sent and the second argument is a Boolean defining whether the data is compressed or not.

The sendDataAction and sendDataFile functions is described below:

```

interface A {

void sendDataAction (string message, bool compression);
void sendDataFile (blob file, bool compression);

};

```

*Listing 3.2 Definition of the interface in Web IDL*

### **void sendDataAction(string message, bool compression)**

Sends arbitrary data through the WebRTC Data channel using the native `RTCDataChannel.send()` method

<b>message</b>	The string to be sent
<b>[compression]</b>	Optional. A Boolean representing whether the data should be compressed before sending true: compresses data using the LZW algorithm false: no data compression (default)

### **void sendDataFile(blob file, bool compression)**

Sends a local file represented by the input File through the WebRTC data channel.

<b>blob</b>	A Blob object representing a file-like object of immutable, raw data to be sent. Blobs represent data that isn't necessarily in a JavaScript-native format.
<b>[compression]</b>	Optional. A Boolean representing whether the data should be compressed before sending true: compresses data using the LZW algorithm false: no data compression (default)

The code of the `sendDataAction` and an example call is shown in the following listing:

```

function sendDataAction(message, compression) {
    if(compression) {
        var data = LZString.compressToUTF16(message);
        data+="C1";
    }
    else data=message+="C0";

    if(sendChannel || receiveChannel) {
        if(isInitiator) sendChannel.send(data);
        else receiveChannel.send(data);
    }
}

// example call
sendDataAction("test message", 0);

```

*Listing 3.3 The sendDataAction function and an example call*

Similarly, the sendDataFile() function uses the sendDataAction() intrinsically to send binary data instead of strings and is also responsible for handling large chunks of data. Sending files is done using the FileReader API which lets web applications asynchronously read the contents of files stored on the user's computer, using File or Blob objects to specify the file or data to read. After the file is read it is split in chunks of 1000 bytes using the slice method.

```

function sendDataFile(file, compression) {
    var reader = new FileReader();
    cmpr=compression;
    reader.readAsDataURL(file);
    reader.onload = onReadAsDataURL;
}

function onReadAsDataURL(event, text) {
    // data object to transmit over data channel
    var data = {};
    // first run
    if (event) text = event.target.result;
    if (text.length > chunkLength) {
        // getting chunk using predefined chunk length
        data.message = text.slice(0, chunkLength);
    } else {
        data.message = text;
        data.last = true;
    }
}

```

```

    sendDataAction("::FILES::"+data.toSource(), cmpr);
    var remainingDataURL = text.slice(data.message.length);
    if (remainingDataURL.length)
        setTimeout(function () {
            onReadAsDataURL(null, remainingDataURL);
        }, 500)
}

```

*Listing 3.4 Sending files over the data channel*

### 3.3 Strings defining actions

The next step is to define a protocol for the exchanged messages. This ensures that the system is well defined and can be easily expanded, but also interoperable so that any WebRTC applications that use this protocol can communicate with each other. The proposed protocol can be used for presenting metadata on video streams, which can include sketching information (Whiteboarding), or chat messaging but can be equally used for any data exchanged between peers including file data (binary), alerts etc.

The way the system communicates actions between peers is done using a very simple language. Two colons (::, Unicode U+003A) are used to indicate that what follows is system data in the form of either strings or “stringified” JSON objects. Chat messages or any other data must be filtered and barred from containing this set of characters. Messages contain an array of information, the elements of which are separated by a double colon (::) (see listing 3.4 for an example message). The first element of the array is always a 5 letter string defining the message type (and in extend the name of the function the system will call, as we will see in the next chapter) e.g.:

- ◆ URMSG: A chat message
- ◆ FILES: An incoming binary file
- ◆ SKTCH: Sketching data etc....

Messages come in two distinct forms: Messages that are intended for canvases and messages that are intended for users. For example a drawing corresponds to a canvas while a chat message corresponds to a user (since a user can have more than one canvas or video shared).

- Messages intended for a canvas or a stream should be in the following form:
   
**::MSGTP::TARGET::[WIDTH]::[DATA]** where
  - MSGTP = Message identification (e.g. SKTCH)
  - TARGET = A string defining which stream this metadata should belong to
  - [WIDTH] = (optional) Width in pixels of the originating video area used for stretching data
  - [DATA] = Stringified JSON Object containing data

- Messages intended for a user should be in this form
   
**::MSGTP::USER::[DATA]** where
  - MSGTP = Message identification (e.g. URMSG)
  - USER = A string defining which user (username) this metadata should belong to
  - [DATA] = Stringified JSON Object containing data (optional)

The above two distinct messages types are shown in the following diagram:

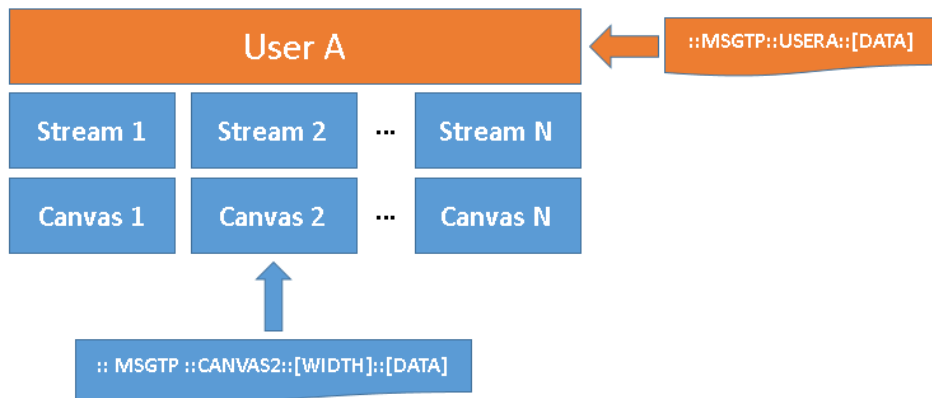


Figure 3.2 System components and sample incoming messages

It must be noted that the system has no means of identifying whether an incoming message is a user message or a canvas message. It is the job of the message handling function to identify the message and act accordingly as we will see in the following section.

In the following table some of the messages that can be exchanged through the data channel and their meaning is explained.

Prefix	Data
<b>::URMSG::UNAME::DATA</b>	A chat message from a user with username “UNAME”
<b>::SKTCH::TARGET::WIDTH::DATA</b>	Sketch data including text caption for the stream named “TARGET”
<b>::FILES::DATA</b>	Data for incoming files
<b>::PAUSE::TARGET::TIME</b>	Pauses a stream at a specified time

*Table 3.1 Sample communication prefixes*

For example to send sketching data (which is sent in the form of a “stringified” JSON object) the following data will be sent:

```

::SKTCH
::pbt6HN5gVideoSketch
::804
::[{"textcaption":""}, {"textcaptionPeer":""}, {"tool":"marker",
color:"#ff0000", size:5, events:[{"x:130, y:458,
event:"mousedown"}, {"x:146, y:447, event:"mousemove"}, {"x:200,
y:401, event:"mousemove"}, {"x:251, y:357, event:"mousemove"},
{"x:439, y:215, event:"mousemove"}, {"x:560, y:121,
event:"mousemove"}, {"x:614, y:80, event:"mousemove"}, {"x:661,
y:47, event:"mousemove"}, {"x:694, y:24, event:"mousemove"},
{"x:711, y:13, event:"mousemove"}, {"x:722, y:5,
event:"mousemove"}, {"x:722, y:4, event:"mousemove"}]]]

```

*Listing 3.5 Sample Sketching Message*

The above message tells peers who receive it that a sketch (SKTCH) must be drawn on the canvas named “pbt6HN5gVideoSketch” which originally has a width of 804 pixels. It can be seen that expandability of the system is very easily achieved by sending prefixed data using the *SendDataAction()* function and then handling them accordingly on the peers who receive them. In the next section we examine how the system handles these messages once they are received.

### 3.4 The handleMessage function

Finally we need to add instructions on how the message should be handled once it is received. This is done in the function *handleMessage*. The function *handleMessage* evaluates incoming messages into function calls. As we saw in the previous chapter messages are comprised of array elements separated by a double colon (::). The first element of the array is always the name of the function to be called while the other elements are parameters of that function. For example when the system receives the string “::MSGNM::PAR1::PAR2::” it will look for a function defined as *msgnm*(*p1*, *p2*) and call it with “PAR1” and “PAR2” as its parameters as shown in the following schematic:

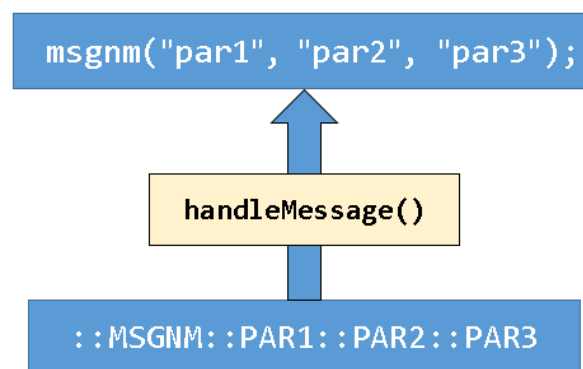


Figure 3.3 Converting messages into function calls

In the following listing we examine the inner workings of the *handleMessage* function which automatically analyses a message and constructs an appropriate function call which it then evaluates. The user of the library implementing the protocol only needs to write a function with the appropriate name and number of parameters.

```
function handleMessage(msg) {  
    var compression= slice(msg 0, -2);  
  
    // do decompression of the message is necessary  
    if(compression=="c1")  
        var event_data=LZString.decompressFromUTF16(msg.data);  
  
    // Split incoming message  
    var splittedMessage= event_data.split("::");
```

```

// How many parameters the function has
var noOfParams=splittedMessage.length - 2;

// What function we should call
var function_name= splittedMessage[1];

// construction of the function call
var fcall= function_name + " (";

for(var i=0;i<noOfParams;i++) {
    fcall=fcall+ ", \"" + splittedMessage[i+2] + "\"";
}

// remove last character (comma)
fcall=slice(fcall 0, -1) + " )";

// The eval function evaluates or executes the argument

eval(fcall);
}

```

*Listing 3.6 Using the handleMessage function*

To demonstrate this functionality, in the next section we show how a new feature can be added to the system with only minimal effort and lines of code.

### 3.5 Example of Expandability

As we explained in previous chapters a developer using our proposed library must set the property RTCdatachannel.onmessage to the provided handleMessage function:

```
datachannel.onmessage = handleMessage;
```

The user of the library must also have the functions sendDataAction and sendDataFile available. Then the system is able to process incoming messages in the way we have explained. To summarize the expandability of the system, we give an example of how we could add a “poke” function that would display a JavaScript alert to the other peer with only 6 lines of code.

Assume an HTML link with the id “pokeLink”:



```
<a href='#' id='pokeLink'>ALERT!</a>
```

First we handle the onclick event of the link to send data using the `sendDataAction` function with the prefix `::POKE::`.

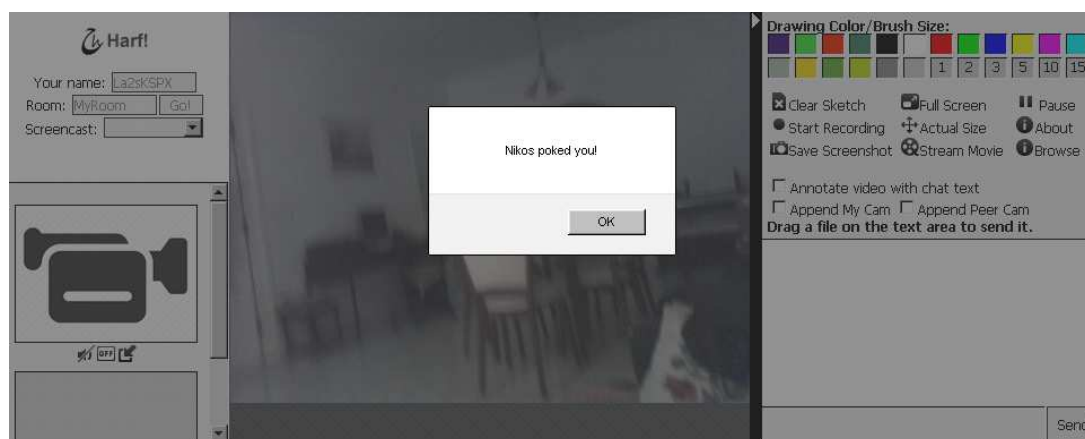
```
pokeLink.onclick=function(){  
    sendDataAction("::POKE::" + username);  
};
```

We then write a function called “POKE” with as many parameters as those defined by the received string (each parameter is separated by the double colon with the first element in the array being the name of the function call):

```
function poke(username) {  
    alert(username + ' poked you!');  
}
```

*Listing 3.7 Calling the `sendDataAction` to send a string and the handling function*

This will result in the following alert message being displayed to the second peer every time the first peer presses the link with id “pokeLink”.



*Figure 3.4 Poke message result*

Similarly other collaborative functions could be integrated into the system with minimal effort.

## Chapter 4. Implementation

In this chapter we discuss in detail the implementation of the WebRTC prototype application we developed, the infrastructure it runs on and the inner workings of the data channel communication.

The application we developed takes advantage of all the APIs described in Chapter 2 and communication model described in Chapter 3. The application is named “Harf” after the Persian word حرف for “talk”. The users have the ability to send video streams to each other. These streams can be sourced from a webcam, an application or window, a monitor, or a local video file. Both users can add text and sketch annotations on any video stream and they can also record it for storage on their local computer. As we discussed in 3.5 where we described the data channel communication protocol of the application, it is very easy for more collaborative features to be added to expand the application.

### 4.1 Infrastructure

As explained in previous chapters, WebRTC requires a minimum load from a server. The server is used once to download the WebRTC application code (in our case, the whole application is less than 160KB including images and code) and a second time to bring the peers together as a signaling server (the data exchanged is no more than a few kilobytes per connection).

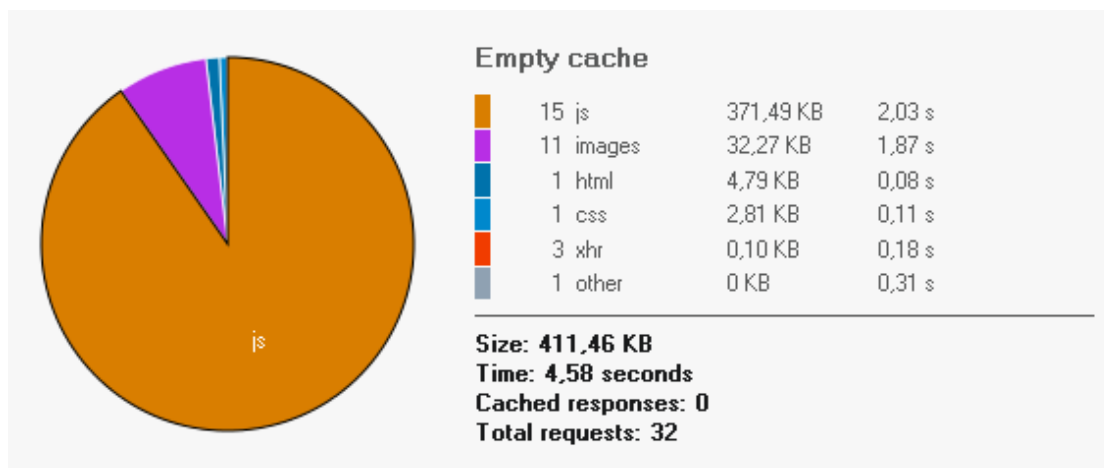


Figure 4.1 Application data per type

In the above figure we see that the whole application downloaded from the server is under 412KB in size including images and external code (jQuery 1.12.3) amounting to a total of 32 HTTP requests. JavaScript amounts to about 90% of the bulk application data. If we take into account the large size of the jQuery library (with a size of almost 234 KB) which is requested from its respective domains and not our application server, we see that the server load for each application pull is about 300KB.

For these reasons we experimented with running both the signaling server and the application host on a single-board computer. The board we selected was the BeagleBone Black which was designed by Texas Instruments.



Figure 4.2 The BeagleBone Black Single Board Computer

BeagleBone was launched in April 2013 and costs about \$45 and uses up to 2W of power, making it a very economical and environmentally friendly solution. The following table shows the hardware specifications of the board:

CPU	Cortex-A8 + 2xPRU(200Mhz)
SOC	AM3358/9
CPU Frequency	1GHz
RAM	512MB DDR3
OS	Debian 8.2 armv7l Linux 4.1.12-ti-r29
Size / Weight	86.40 mm × 53.3 mm / 40g

Table 4.1 BeagleBoard Black Specifications

The system runs a precompiled distribution of Node.js v.0.10.41 for the BeagleBoard Black [114]. The Node.js server is then run using “forever”, a simple CLI tool for ensuring that a given script runs continuously

```
root@beaglebone:~/Server# forever start serveHarf.js
warn:    --minUptime not set. Defaulting to: 1000ms
warn:    --spinSleepTime not set. Your script will exit if it
does not stay up for at least 1000ms
info:    Forever processing file: serveHarf.js
```

*Listing 4.1 Starting the Node.js server script using Forever*

## 4.2 Implementation of the Signaling Server

The job of the signaling server is to listen for messages and broadcast them to potential WebRTC peers. In this section we explain how the signaling server operates, in more detail.

Currently WebRTC screen sharing works only behind SSL enabled web servers on Firefox. For this reason the Node.js HTTPS module is used. HTTPS is the HTTP protocol over TLS/SSL. In Node.js this is implemented as a separate module, its use shown in the following listing:

```
var static = require('node-static');
var https = require('https');
var file = new(static.Server)();
var fs = require('fs');

var hskey = fs.readFileSync('harf-key.pem');
var hscert = fs.readFileSync('harf-cert.pem');

var options = {
    key: hskey,
    cert: hscert
};

var app=https.createServer(options, function (req, res) {
    file.serve(req, res);
}).listen(443);
```

*Listing 4.2 Creating an SSL server in Node.js*

Initially the server loads the following Node.js modules:

- ◆ Node-static: An RFC 2616 compliant HTTP static-file server module.
- ◆ HTTPS: HTTP protocol over TLS/SSL.
- ◆ fs: File I/O module.

The fs module is then used to read two PEM certificates on the web server. The first file is *harf-key.pem* and contains the private key used for SSL, the second file is *harf-cert.pem* which contains the public x509 certificate to use. PEM is a container format defined in RFCs 1421 to 1424 that can include public certificates or certificate chains including public keys, private keys and root certificates. The contents of these two PEM files are fed into the `createServer` method of the HTTPS module to create a static server that listens to port 443 (standard HTTPS port).

Once the server is running, Node.js loads the main modules used for the signaling server: Socket.io. The module simply waits for messages from clients. Messages can be of the following two types:

- ◆ **Create or Join** message: This message is sent by clients that wish to either create a room or join an existing one. The server automatically creates a room with the requested name if it doesn't exist or tries to join to it if it exists. The server then answers back with 'created' if a room was created, 'joined' if the client successfully joined an existing room or 'full' if the room did exist but it would not accept any more peers.

```
if (numClients == 0){
  socket.join(room);
  socket.emit('created', room, username);
}
```

*Listing 4.3 Creating a socket.io room on the signaling server*

- ◆ **Message**: When the server receives a 'message' it simply broadcasts it to all the other peers in the room. This is used to broadcast exchange signaling information between the peers that will be used to establish a WebRTC P2P session.

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

*Listing 4.4 Broadcasting messages from the signaling server*

### 4.3 The Client Application

The source code of the client application is structured in the following simple way:

```
+---Server
+---serverHarf.js
+---index.html
+---main.css
\---images
\---js
    +---adapter.js
    +---canvas.js
    +---clientHarf.js
    +---cobrowsing.js
    +---file.js
    +---html.js
    +---lz-string.min.js
    +---recording.js
    +---screencast.js
    +---sender.js
    +---sketch.js
    +---utils.js
```

Table 4.2 List of application files

The purpose of each file is described below:

- ◆ **serveHarf.js:** The signaling server Node.js script
- ◆ **index.html, main.css:** A single HTML file containing the interface layout and a CSS file describing the interface style
- ◆ **images directory:** Contains all the graphics (buttons etc.) of the application
- ◆ **adapter.js:** The WebRTC interoperability library
- ◆ **canvas.js:** Contains function for manipulating canvases
- ◆ **clientHarf.js:** The main WebRTC connectivity library, contains function for communicating with the signaling server and establishing a peer to peer connection.
- ◆ **cobrowsing.js:** Functions for basic co-browsing
- ◆ **file.js:** Functions for reading and streaming local video files
- ◆ **html.js:** Functions for dynamically outputting HTML code
- ◆ **lz-string.js:** Functions for compressing strings

- ◆ **recording.js:** Functions for recording video streams and storing them as webm files.
- ◆ **screencast.js:** Functions for the WebRTC screen capturing API
- ◆ **sender.js:** Functions for sending local files through the WebRTC data channel
- ◆ **sketch.js:** Contains the whiteboard drawing functionality
- ◆ **utils.js:** General utility functions

## 4.4 The Interface

The interface of the developed application consists of the following areas:

1. The connection box
2. The streams list
3. The maximized stream area
4. The toolbox
5. The chat area

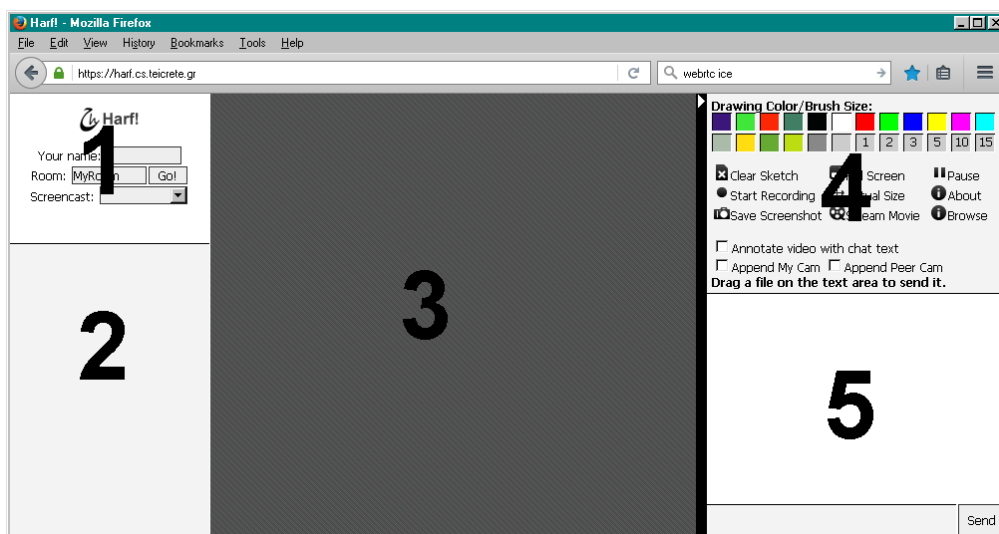


Figure 4.3 The application interface

The connection box is where the users can enter a username and a room name. Users can also choose to start a screen sharing session from using the dropdown box in the area.

The streams list is a scrollable area where all the streams available to the users are shown. Each stream is displayed as a thumbnail video with buttons representing actions below it, as shown in the following figure.



Figure 4.4 Sample stream thumbnails with available actions. From left to right: window, webcam and a local video file

To enable streaming of local media we used the experimental stream capturing API currently implemented in Mozilla Firefox as shown in the following listing:

```
localstream=getel(localvideo).mozCaptureStream();




pc.onnegotiationneeded = function (event) {
    pc.createOffer(setLocalAndSendMessage,
        onSignalingError, sdpConstraints);};

function sendMovieStream() {
    pc.addStream(localstream);
}
```

Listing 4.5 Using the capturestream method to capture a stream from a video element

The captureStream() method produces a real-time capture of the media that is rendered to the media element and is defined in W3C’s “Media Capture from DOM Elements” working draft [115].

The buttons below the thumbnail vary depending on the type of the stream and these include:

-  Blanks or reveals the video stream
-  Mutes or unmutes the audio stream
-  Maximizes the stream bringing on the center area of the page, or minimizes it hiding it from the center area of the page





Attempts to send the stream over the `RTCPeerConnection` to the other peers.

Clicking on the maximize button brings the selected stream on the center area of the page and enables the collaborative controls for this specific stream.

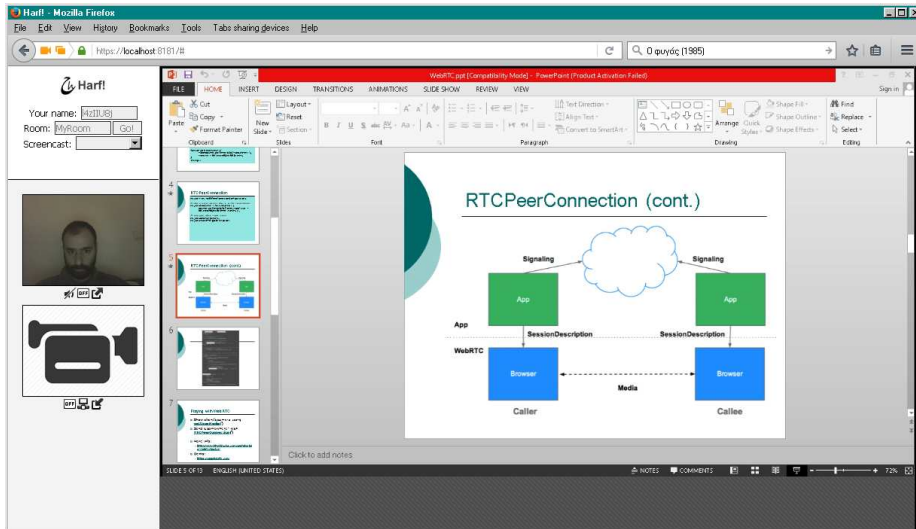


Figure 4.5 A screen capturing session with the PowerPoint window maximized

Once connection is established users have a range of tools available from the sidebar on the right side of the screen. The most important feature is the whiteboard toolbar from which the user can select an ink color and a brush size.

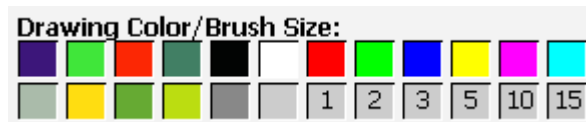










Figure 4.6 The whiteboard toolbar

Below the whiteboard toolbar are the other collaborative options available to users. These options are:

-  Starts or stops recording the currently maximized stream
-  Clears all sketches on the currently maximized stream
-  Pauses or freezes the currently maximized stream
-  Prompts the user to select a local video file which can then be streamed to the other peers

-  Captures a single frame from the currently maximized stream which can then be saved as a PNG file.
-  Shows the currently maximized stream in full screen. All annotations options are disabled in this mode and users cannot annotate the stream.
-  Resized the currently maximized stream to its original size.
-  Replaces the currently maximized stream with a HTML document hosted on the same domain as the WebRTC application. Basic co-browsing is offered to the peers in this mode.

Users can communicate using text messages by typing something in the text area on the bottom-right corner of the screen. To demonstrate the use of the communication protocol, once a user clicks the send button the system constructs a URMSG message in the following manner:

```
// getting the text typed by the user and filtering it from
// possibly malicious elements or the :: sequence
var data = clearString(sendTextarea.value);

// construct an appropriate message
var datamsg = "::URMSG::"+username+"::"+data;

// send the message without compression
sendDataAction(datamsg, 0);
```

*Listing 4.6 Constructing the chat message*

Now upon receiving the above message the system will call the URMSG function as following:

```
function urmsg(username, text) {

var peerusername;

if(username=="") peerusername="Peer";
else peerusername=username;
```

```

// also display the received text on the active video stream
// if user has enabled this option.
if(getel("annotateVideoOption").checked) {
sketches[elementNameG+"Sketch"].sketch().actions[0].textcaptionPeer=
text;
sketches[elementNameG+"Sketch"].sketch().redraw();
}

// write message on chat area
receiveTextarea.insertAdjacentHTML('beforeEnd',
"<greyed style='color:#555555'>" + peerusername + ", " +
hrDate() + ": </greyed>" + text + "<br/>");

// scroll down the chat area
receiveTextarea.scrollTop = receiveTextarea.scrollHeight;
}

```

*Listing 4.7 The Urmsg function*

The application also communicates messages to the user using the chat area below the toolbox. For example when a user clicks on the “Take screenshot” button the system sends a message to the user notifying him of the link from which he can download it. This increases system usability by eliminating the use of popups or other types of alerts. These “system messages” are only visible to the user they concern and not to the other peers of the session.

```

George, 19:30: Hello
Nick, 19:30: Hey how are you?
System, 19:30: You took a screenshot! Click here to
download it!

```

*Figure 4.7 Example system messages*

The chat area can also be used for exchanging files between users. A user can drag and drop a file on the text area to send it to other users. In chapter 3 we discussed the function `sendDataFile` function that can send local files to other peer. The file function that handles incoming file data is shown below:

```

function files(event_data) {
  var data = eval(event_data);
  arrayToStoreChunks.push(data.message);
  if (data.last) {
    // returns blob URL of incoming file
    return arrayToStoreChunks.join('');
  }
}

```

```

arrayToStoreChunks = []; // resetting array
    }
}

```

Listing 4.8 The file function

The result is shown in the following screenshot:

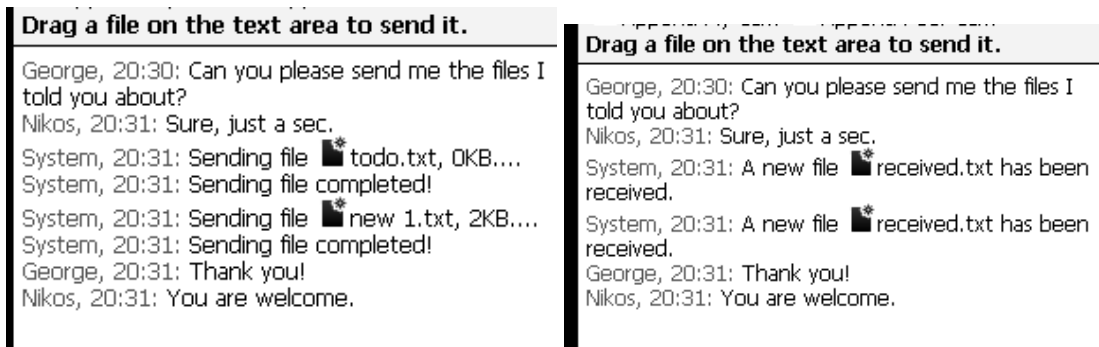


Figure 4.8 Users exchanging files

Every stream has its own whiteboard data attached to it. Users can draw on any surface that is maximized and the data is sent through the WebRTC data channel to the other peer.

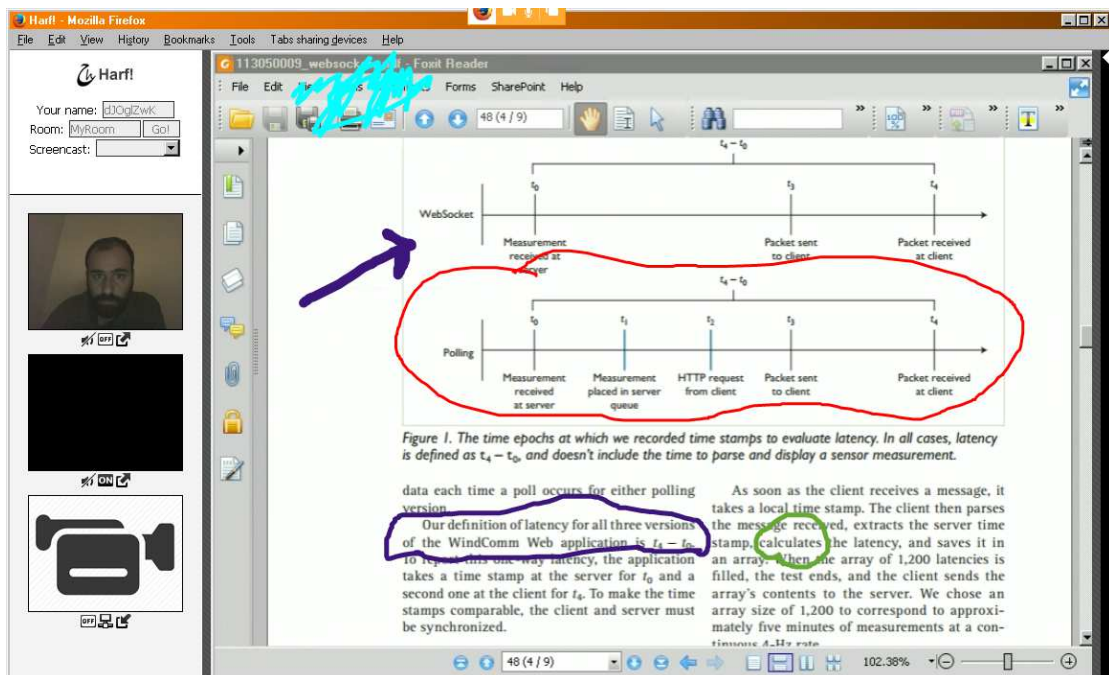


Figure 4.9 Users sketching on a PDF document

Using the MediaRecording API described in section 2.3.1 we can achieve recording of every canvas and the sketching or other action in a WebM file. Unfortunately it is impossible to capture audio on the OS level (e.g. audio played by an application) because WebRTC does not have access to it yet. What we can capture is audio from streaming media files and user web cams. To do that we use the HTML5 MediaRecorder API described in section 2.3.1. In the following screenshot a streaming movie is captured along with all the sketching action users draw on it. Once the user clicks the “Stop Recording” button the movie is made available to him in the form of a webm blob which can be downloaded and played locally in his computer.

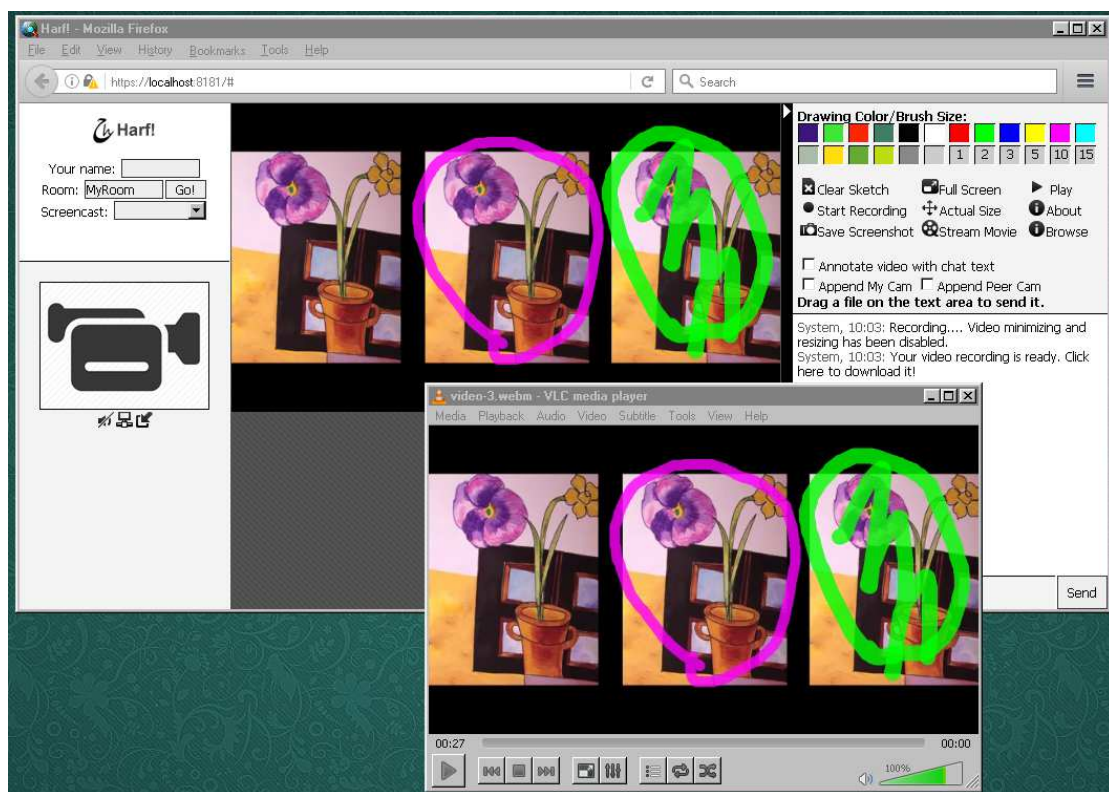


Figure 4.10 Recording and then playing a WebM file locally

To achieve this a combination of the stream capture and the media recorder API are used. Initially we create a media recorder object whose input is a captured stream from the currently active canvas:

```
var tempCanvasStream=getel('tempCanvas').captureStream();  
mediaRecorder = new MediaRecorder(tempCanvasStream);
```

When the user clicks the start recording button we get the audio stream from the video element if it exists and add it on the temp canvas. This is done in order to be possible to also record audio:

```
if(streams[streamNameG].getAudioTracks()[0]!=undefined) {  
tempCanvasStream.addTrack(streams[streamNameG].getAudioTracks(  
)[0]);  
}
```

Finally when the recording is stopped by the user the audio stream on the canvas is disabled and the system waits for the ondataavailable callback function:

```
mediaRecorder.ondataavailable = function(e) {  
var videoURL=window.URL.createObjectURL(e.data);  
}
```

Another feature of the system is the ability to draw text annotation on each stream by selecting the appropriate checkbox in the toolbox area.

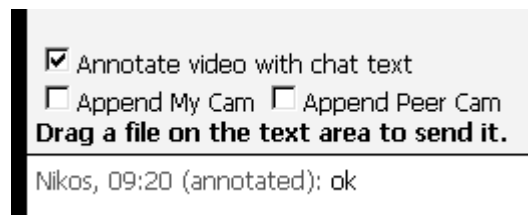


Figure 4.11 Selecting to embed incoming chat messages on the vide ostream

Once a user makes this selection all messages, incoming and outgoing are appended on the currently maximized video stream.

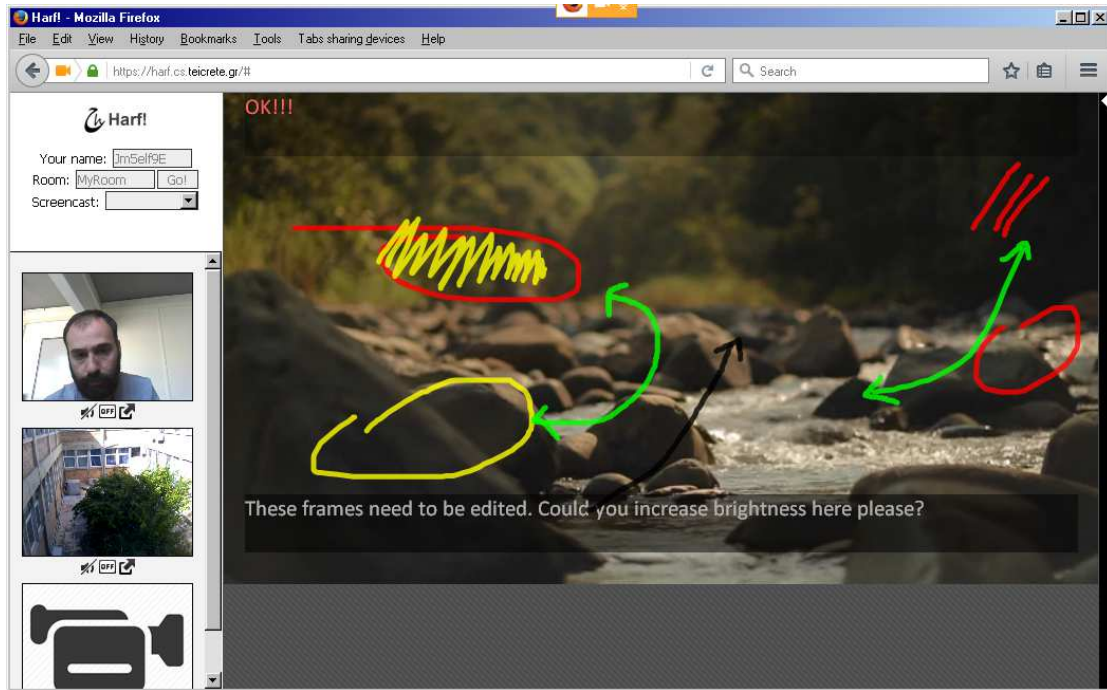


Figure 4.12 Adding text annotation on a shared webm video

Sketching data are in the form of JSON object which are comprised of action and mouse coordinates.

```
[{
  textcaption: "Test!"
}, {
  textcaptionPeer: ""
}, {
  tool: "marker",
  color: "#f00",
  size: 5,
  events: [{
    x: 162.5,
    y: 125,
    event: "mousedown"
  }, {
    x: 162.5,
    y: 124,
    event: "mousemove"
  }]
}]
```

Listing 4.9 Sample sketching actions JSON object

The JSON object comprises of 3 values: A value named “textcaption” which contains the text caption of the local user, “textCaptionPeer” which contains the text caption of the remote peer and then it contains a list of mouse action and sketching data. For example the object of the previous listing will result in a caption “Test!” and a red dot at coordinates 162.5, 124 as seen in the following screenshot:

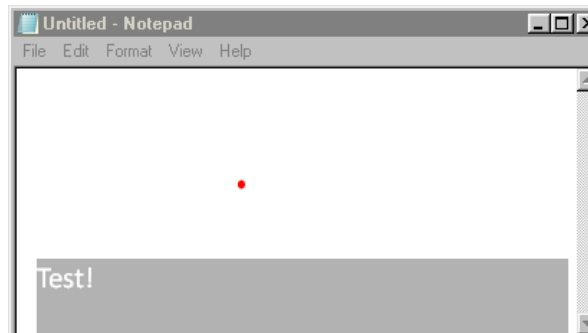


Figure 4.13 Resulting sketch

The JSON object is converted to text and then compressed using UTF16 LZW compression before sent through the WebRTC data channel.

```
␣欸溢ε○→␣␣竹氩忽J␣樹␣␣␣3訖␣␣|z½底嶼携妃,繪␣␣␣容␣滿嶼␣␣=缺呂哈榛K,قWb␣␣イ鼻␣␣␣R9dnā瞞繫揭3斤焯  
秘N␣␣男刮齋※~␣␣F矜榛␣盒瞞窳案␣嬋␣␣␣倣≡嗜␣␣×␣␣␣絨芻戩
```

Listing 4.10 Unicode characters resulting from compressing a string to UTF16 LZW

Referring to table 3.1 we see that sketch messages come in the following form:

<b>::SKTCH::TARGET::WIDTH::DATA</b>	Sketch data including text caption for the stream named “TARGET”
-------------------------------------	--

In accordance with the protocol upon receiving a sketch message (that is a message prefixed with the “::SKTCH::” string the system will call the sktch function:

```
function sktch(target, width, data) {
  // what is the width of our own target canvas
  var my_sktch_width=getel(target_sketch).width;

  // calculate ration needed to resize sketch
  var width_ratio=my_sktch_width/width;

  // convert sketch data back to a JSON object and replace
  // the target canvas sketch data with it
  sketches[target_sketch].sketch().actions=eval(data);
}
```



```
// if incoming data is not empty apply transformation
if(data!="[]")
// sketchTransform function included a call to the redraw
// function
sketchTransform(target, width_ratio);
else
// redraw sketch (in this case it means erase sketch)
sketches[target_sketch].sketch().redraw();
}
```

*Listing 4.11 The sketch function*

## Chapter 5. Benchmarking

### 5.1 Compression Efficiency

To measure compression efficiency we connected two computers and measured the time it took to render a sketch depending on the size of the JSON object that described the sketch.

The LZW compression algorithm is very efficient for sketching metadata because of the high number of keywords and word iterations. The following screenshots represents about 100KB of sketch data drawn on the Notepad window. When the data is compressed using the LZW algorithm the resulting data is 6.6KB a compression ratio of 93%.

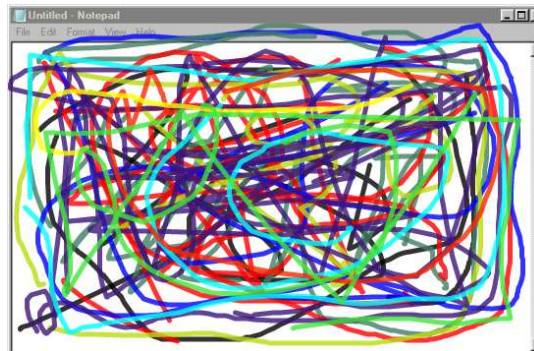


Figure 5.1 103KB of uncompressed sketch data (6.6 KB of transferred data using compression)

It is obvious that using compression on the data channel can dramatically decrease network overhead and with modern hardware the compression/decompression times on the local host system are actually miniscule as shown in the following table

Test No.	Bytes Before Decompression	Bytes After Decompression	Compression Ratio (%)	Relative Time Difference (ms)
1	235	1001	77	26
2	486	3044	84	32
3	2001	21645	91	78
4	2668	31585	92	107
5	3255	40661	92	101
6	3991	51650	92	99
7	5728	76730	93	196
8	7535	104437	93	243

Table 5.1 Compression Efficiency

In the above table we measure the time required to render a graphic on the canvas assuming that the initial rendering (empty canvas) requires 0 time. Rendering time is measured from the time a user makes a sketch to the time it is rendered on the other peer's computer. We repeated the process 8 times with increasing size of transferred data.

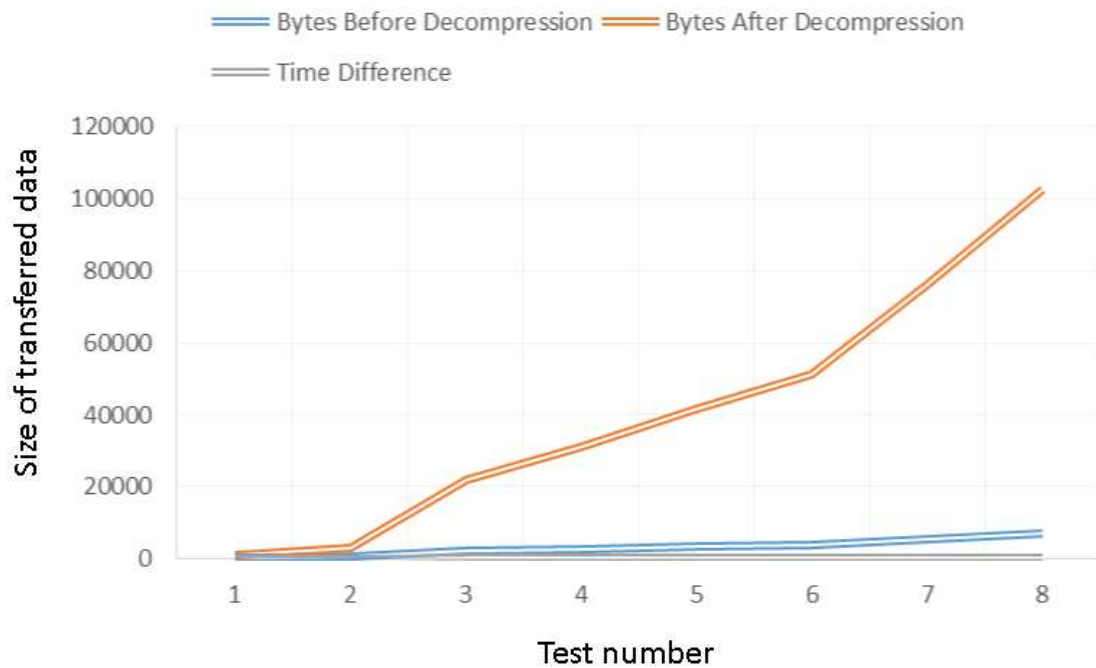


Figure 5.2 Compression Efficiency

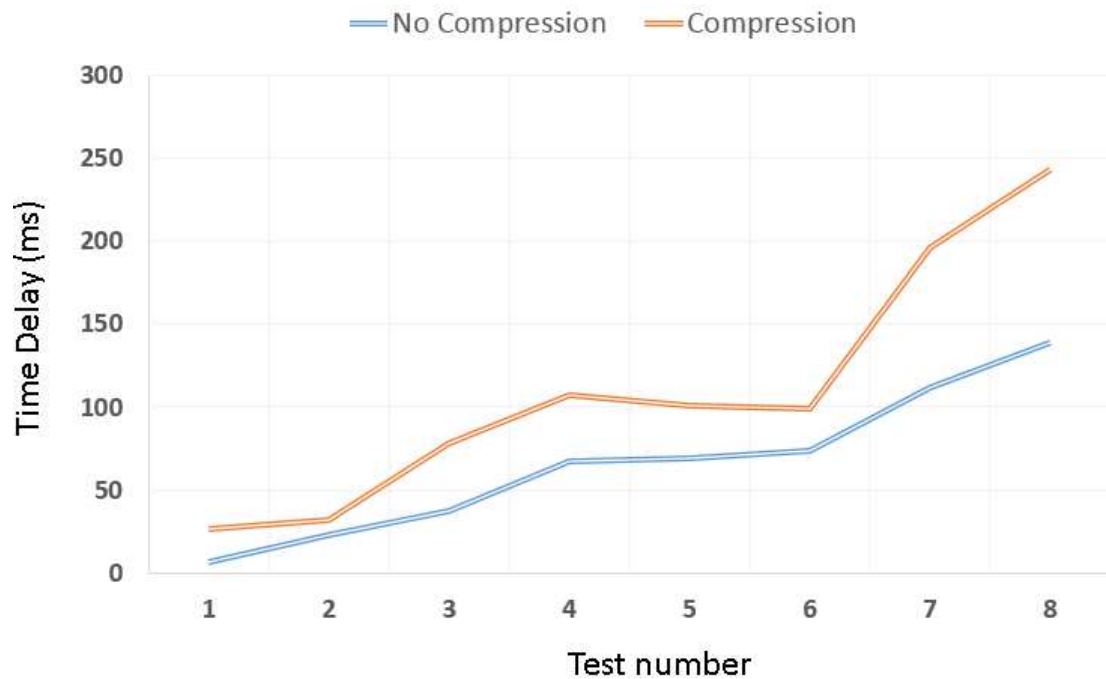
We repeated the same 8 test but this time without compression on the data channel.

**Without compression:**

Test No.	Bytes Transferred	Relative Time Difference (ms)
1	1071	6
2	3238	23
3	20361	37
4	31029	67
5	41590	69
6	52648	74
7	76359	112
8	103789	139

Table 5.2 Data transfer rates without compression

The above data are presented in the following chart:



5.3 Comparison between compression and no compression on a LAN

We observe that compressing the data exchanged via a WebRTC data channel over a LAN connection (or other networking situations where network bandwidth is ample) may not be an optimal choice, since when comparing the time required compressing a string with the time required to decompress it the overhead is significant. Nevertheless the very high compression ratio achieved by the compression algorithm which is up to 93% for this type of data may prove to be useful in situations where network infrastructure is limited or expensive.

## 5.2 CPU Consumption

On the signaling server, a typical session description message is about 2KB in size, while a candidate offer message is about 150 bytes. Assuming that each peer exchanges one session description message and 5 candidates on average, we see that for each peer connection, less than 5 kilobytes of data (10 kilobytes for 2 peers) are sent and received by the signaling server.

On the modern client devices, hardware is powerful enough for all the video and canvas operations that are required by most applications including our own. Furthermore, HTML5 Hardware Accelerated canvas are implemented on most platforms and browsers taking advantage of the capabilities of modern GPUs.

GPU #2 Active	false
GPU Accelerated Windows	1/1 Direct3D 11 (DMTC)
Subsys ID	21c517aa
Supports Hardware H264 Decoding	Yes
Vendor ID	0x8086
WebGL Renderer	Google Inc. -- ANGLE (Intel(R) HD Graphics Direct3D9Ex vs_3_0 ps_3_0)
windowLayerManagerRemote	true
AzureCanvasBackend	skia
AzureContentBackend	cairo
AzureFallbackCanvasBackend	cairo
AzureSkiaAccelerated	0

Table 5.3 Firefox configuration information indicating hardware acceleration using the Windows Direct3D

To measure the processing power requirements of our system we used the embedded developer tools in Mozilla Firefox 46. The test system was a laptop equipped with 4GB of RAM and an Intel Core i3 (U38) CPU with a clock speed of 1.33 GHz. The computer was running the Microsoft Windows 7 64bit operating system. The system can be considered outdated by today's standards.

To analyze which processes consume more time we used the Firefox Performance Tool and conducted two 20 second tests: During the first test the system was used for streaming media between two peers while during the second test the sketching feature was also used. During the first test the average frame rate was measured at 42 fps while during the second test the average frame rate was at 17 fps.

Function	Function Cost	
	Sketching & Streaming	Streaming
Gecko (includes idle time)*	37.06%	60.83%
Sketching	19.06%	-
Graphics*	16.40%	23.57%
Garbage Collecting*	9.41%	6.52%
JIT*	4.83%	1.88%
Tools*	2.25%	3.28%
Input & Events*	1.79%	-
Compression/Decompression Algorithms	1.71%	-
Other	7.49%	3.92%

\* Denotes internal browser functions

Table 5.4 Function costs while sketching and while straming

In the following figure we see the frame rate in which the browser renders the page during the tests.



Figure 5.4 Comparison of frame rate during simple streaming (above) and during sketching (below). Negative spikes denote that the user is sketching on the canvas.

Although we observe a drop in the framerate during sketching with the test system, with modern CPUs, very high framerates throughout the operation are achieved making this drop unnoticeable by users.

## Chapter 6. Conclusions & Future Work

In this thesis we presented an extensive review of the WebRTC technology and its possible application in the fields of synchronous online collaboration, screen sharing, and peer to peer media streaming in the browser.

The protocol we describe in section 3.5 is the foundation of our application and is based on an assumption that a WebRTC application is comprised of users and streams. Because streams in HTML5 boil down to video elements we assign an HTML canvas to each stream. We then define a message protocol that uses the WebRTC data channel and can exchange messages that can have an impact on either individual users (chat messages, file exchanges, alerts) or canvases (drawings, annotations etc).

The application we developed is a prototype intended to demonstrate the capabilities of WebRTC and its potential use for online collaboration and media streaming. As such it has in itself many “bugs” and is missing some of the features that would be normally encountered in a commercial product. Apart from the occasional bug, some features that could be further explored and implemented include a co-browsing feature which utilizing the data channel synchronizes mouse movements and URLs in a host web site.

As it is now the system only supports two peers with equal privileges in each room. Another approach is a “one-to-many” model where only one user with elevated privileges will have the ability to share streams while all other users in the room will only be able to use the data channel to collaborate but without the ability to add streams on their own. In this approach the user with the elevated privileges could also have “floor-management” option at his disposal (eg. adding/removing users from the room, selecting which collaborative functions each user has available etc)

WebRTC is a work in progress. Many of the APIs used in this thesis are still in development and implementation. As a result some of the features of WebRTC used in this thesis are either not implemented in all major browsers or have some bugs. Most notable implementation bugs are the Mozilla bugs 1178751 and 912907 we discussed in section 2.3.1. For this reason we decided to focus our development on the Mozilla Firefox browser which as in April 2016 has almost all WebRTC proposed features implemented. As more commercial WebRTC applications are more people are using it, new browser versions are expected to have more stable implementations

of the WebRTC APIs. The WebRTC working group meets on an ad-hoc basis week on the phone and using mailing lists.

Furthermore it must be noted that even the core technologies used in WebRTC are subject to change. For example while the IETF decided mandatory video codecs to be H.264 and VP8 as we discussed in section 2.2.5, Firefox and Chrome versions released in April and May 2016 also included the VP9 codec. With VP9, internet connections that are currently able to serve 720p without packet loss or delay will be able to support a 1080p video call at the same bandwidth. VP9 can also reduce data usage for users with poor connections or expensive data plans, requiring in best cases only 40% of the bitrate of VP8 [116].

What the future holds for WebRTC is not easy to say. Many argue that with “peak telephony” (the point in time when telephony communication was at its maximum) reached in the United States, the UK and other countries, RTC technologies have lost the battle with asynchronous forms of communication like email, social networks and IM. Technologies like WebRTC are helping to change the definition of RTC. Telephony used to be constrained by an apparatus known as a phone. Today the medium of RTC continues to change as application and web sites add RTC as a feature (e.g. Facebook Messenger, WhatsApp, SnapChat etc.) [117]. In the end WebRTC is a technology used to enhance a service or application, changing the fundamental model of communications by creating a world where anyone has the ability to put communications in their applications without resorting to a communications company.



## References

- [1] V. Pimentel and B. G. Nickerson, "Communicating and Displaying Real-Time Data with WebSocket," *IEEE Internet Computing*, vol. 16, no. 4, pp. 45 - 53 , 2012.
- [2] T. Levent-Levi, "4 Facts You Need to Know about P2P in WebRTC," BlogGeek.me, 30 9 2013. [Online]. Available: <https://bloggeek.me/4-p2p-webrtc-facts/>.
- [3] Apache Cordova, "Architectural Overview of Cordove Platform," [Online]. Available: <http://cordova.apache.org/docs/en/latest/guide/overview/>. [Accessed 11 4 2016].
- [4] R. O. B. C. R. R. J. Jay F Nunamaker Jr, *Collaboration Systems: Concept, Value, and Use*, New York: Routledge, 2014.
- [5] C. A. Ellis, S. J. Gibbs and G. Rein, "Groupware: some issues and experiences," *Communications of the ACM*, vol. 34, no. 1, pp. 39-59, 1991.
- [6] T. Walhert, "Synchronous or Asynchronous Tools," Green Hills Area Education Agency, [Online]. Available: <https://sites.google.com/a/ghaea.org/aiw-iowacore-techintegration/synchronous-vs-asynchronous>. [Accessed 16 4 2016].
- [7] B. Kask and S. Wood, "Synchronous and Asynchronous Communication:Tools for Collaboration," University of British Columbia, [Online]. Available: [http://etec.ctlt.ubc.ca/510wiki/Synchronous\\_and\\_Asynchronous\\_Communication:Tools\\_for\\_Collaboration](http://etec.ctlt.ubc.ca/510wiki/Synchronous_and_Asynchronous_Communication:Tools_for_Collaboration). [Accessed 16 4 2016].
- [8] H. J. Smith, S. Higgins, K. Wall and J. Miller, "Interactive whiteboards: boon or bandwagon? A critical review of the literature," *Journal of Computer Assisted Learning*, vol. 21, no. 2, pp. 91-101, 2005.
- [9] C. J. Wenning, "Whiteboarding & Socratic dialogues: Questions & answers," *Journal of Physics Teacher Education Online*, vol. 3, no. 10, pp. 3-10, 2005.
- [10] O. Akbaş, M. Baturay and a. Y. Söker, "Interactive Whiteboard-Based ATM Use Training for Older Individuals," *International Online Journal of Educational Sciences*, vol. 8, no. 1, pp. 87-97, 2016.
- [11] H. M. Abdel-Wahab and M. A. Feit, "XTV: a framework for sharing X Window clients in remote synchronous collaboration," in *IEEE Conference on Communications Software*, Chapel Hill, NC, 1991.
- [12] C. A. Jara, F. A. Candelas, F. Torres, C. Salzmann, D. Gillet, F. Esquembre and S. Dormido, "Synchronous collaboration between auto-generated WebGL applications and 3D virtual laboratories created with Easy Java Simulations," in *9th IFAC Symposium Advances in Control Education*, Nizhny Novgorod, 2013.
- [13] Z.-E. Andrioti, *Web3D Gaming Over HTML5 and Web-Based Communication*, 2015.
- [14] K. Kapetanakis, H. Andrioti, H. Vonorta, M. Zotos, N. Tsigkos and P. I, "Collaboration framework in the EViE-m platform," in *Proceedings of the 24th EAEEIE Annual Conference*, 2013.
- [15] S. M.-V. Metz, P. Marin and E. Vayre, "The shared online whiteboard: An assistance tool to synchronous collaborative design," *European Review of Applied Psychology*, vol. 65, no. 5, pp. 253-269, 2014.

- [16] J. Franke and B. Cheng, "Real-time privacy-preserving cobrowsing with element masking," in *2013 17th International Conference on Intelligence in Next Generation Networks (ICIN)*, Venice, 2013.
- [17] M. Rouse, "Collaborative browsing (co-browsing)," [Online]. Available: <http://searchcrm.techtarget.com/definition/collaborative-browsing>. [Accessed 26 3 2016].
- [18] Oracle, "Best Practices for Oracle RightNow Cobrowse Cloud Service," 3 2012. [Online]. Available: <https://www.oracle.com/applications/customer-experience/rightnow/web-experience/standalone-cobrowse/index.html>.
- [19] E. S. Crandall, A. Fernstedt, S. L. Greenspan and D. M. Weimer, "Method and apparatus for internet co-browsing over cable television and controlled through computer telephony". United States Patent US 6425131 B2, 23 7 2002.
- [20] A. Roy, D. R. Prabhu, J. R. Doering, X. Kuang and R. Sunkara, "System and method for real-time co-browsing". United States Patent US 7149776 B1, 12 12 2006.
- [21] I. Paul, "Video, audio streaming gobble up 70% of peak Internet traffic in North America," PCWorld, 8 12 2015. [Online]. Available: <http://www.pcworld.com/article/3012625/streaming-services/video-audio-streaming-gobble-up-70-of-peak-internet-traffic-in-north-america.html>.
- [22] Z. Shen, J. Luo, R. Zimmermann and A. V. Vasilakos, "Peer-to-Peer Media Streaming: Insights and New Developments," *Proceedings of the IEEE*, vol. 99, no. 12, pp. 2089-2109, 2011.
- [23] P. Lubbers and F. Greco, "HTML5 WebSocket: A Quantum Leap in Scalability for the Web," [Online]. Available: <http://www.websocket.org/quantum.html>. [Accessed 22 3 2016].
- [24] Mozilla Developer Network, "WebSockets," [Online]. Available: [https://developer.mozilla.org/en-US/docs/Web/API/WebSockets\\_API](https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API). [Accessed 5 3 2016].
- [25] V. Wang, F. Salim and P. Moskovits, "The WebSocket API," in *The Definitive Guide to HTML5 WebSocket*, Apress, 2013, pp. 13-32.
- [26] M. Ubl and E. Kitamura, "Introducing WebSockets: Bringing Sockets to the Web," 49 10 2010. [Online]. Available: <http://www.html5rocks.com/en/tutorials/websockets/basics/>.
- [27] J. Freeman, "9 killer uses for WebSockets," 14 11 2013. [Online]. Available: <http://www.javaworld.com/article/2071232/java-app-dev/9-killer-uses-for-websockets.html>.
- [28] "HTTP/2 Frequently Asked Questions," [Online]. Available: <https://http2.github.io/faq>. [Accessed 22 3 2015].
- [29] Y. Hirano, "WebSocket over HTTP/2.0," 14 2 2014. [Online]. Available: <http://tools.ietf.org/html/draft-hirano-httpbis-websocket-over-http2-00#section-1>.
- [30] "WebRTC," [Online]. Available: <http://www.webrtc.org/home>. [Accessed 4 7 2015].
- [31] "Is WebRTC ready yet?," [Online]. Available: <http://iswebrtcreadyet.com/>. [Accessed 2 3 2016].
- [32] J. Wagner, "What Developers Should Know About ORTC Versus WebRTC," ProgrammableWeb, 12 10 2015. [Online]. Available: <http://www.programmableweb.com/news/what-developers-should-know-about-ortc-versus-webrtc/analysis/2015/10/12>.
- [33] ABI Research, "4.7 Billion Mobile WebRTC Devices by 2018 Despite Lack of Open

- Support from Apple and Microsoft," 25 9 2013. [Online]. Available: <https://www.abiresearch.com/press/47-billion-mobile-webrtc-devices-by-2018-despite-l/>.
- [34] Google Chrome team, "Interop Notes," Google Inc., [Online]. Available: <http://www.webrtc.org/web-apis/interop>. [Accessed 27 9 2015].
- [35] S. Anderson, "Battle of the Codecs: Is VP8 or H.264 Better for WebRTC?," WebRTC World, 11 11 2013. [Online]. Available: <http://www.webrtcworld.com/topics/webrtc-world/articles/359800-battle-the-codecs-vp8-h264-better-webrtc.htm>.
- [36] A. Roach, "WebRTC Video Processing and Codec Requirements," 12 6 2015. [Online]. Available: <https://tools.ietf.org/html/draft-ietf-rtcweb-video-06>.
- [37] J. Valin and C. Bran, "WebRTC Audio Codec and Processing Requirements," 9 2 2016. [Online]. Available: <https://tools.ietf.org/html/draft-ietf-rtcweb-audio-10>.
- [38] D. Mohny, "What Will We Do with Billions and Billions of WebRTC Devices?," Real Time Communications, 22 4 2015. [Online]. Available: <http://www.realtimerecommunicationsworld.com/topics/realtimerecommunicationsworld/articles/402077-what-will-we-with-billions-billions-webrtc-devices.htm>.
- [39] Ericsson Research, "Browser," OpenWebRTC, [Online]. Available: <http://www.openwebrtc.org/browser/>. [Accessed 3 3 2016].
- [40] Mozilla, "Firefox Hello," [Online]. Available: <https://www.mozilla.org/en-US/firefox/hello/>. [Accessed 7 3 2016].
- [41] Zingaya, [Online]. Available: <https://zingaya.com/>. [Accessed 7 3 2016].
- [42] K. Jain, A. Himmatramka, A. Bhandary, A. D'silva and D. Barge, "Synchronized Development Using WebRTC Real-Time Collaboration in WebRTC," *International Journal of Engineering Science*, vol. 6, no. 4, 2016.
- [43] I. V. Osipov, A. A. Volinsky and A. Y. Pratikova, "E-Learning Collaborative System for Practicing Foreign Languages with Native Speakers," *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 3, 2016.
- [44] C. Cola, Cluj-Napoca and H. Valean, "E-health appointment solution, a web based approach," in *E-Health and Bioengineering Conference (EHB)*, Iași, 2015.
- [45] A. P. Vidul, S. Hari, K. P. Pranave, K. J. Vysakh and K. R. Archana, "Telemedicine for emergency care management using WebRTC," in *International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Kochi, 2015.
- [46] J. Jang-Jaccard, S. Nepal, B. Celler and B. Yan, "WebRTC-based video conferencing service for telehealth," *Computing*, vol. 88, no. 1, pp. 169-193, 2016.
- [47] L. V. Ma, J. Kim, S. Park, J. Kim and J. Jang, "An efficient Session\_Weight load balancing and scheduling methodology for high-quality telehealth care service based on WebRTC," *The Journal of Supercomputing*, pp. 1-18, 2016.
- [48] "Sharefest," [Online]. Available: <https://www.sharefest.me/faq>. [Accessed 7 3 2016].
- [49] "WebTorrent," [Online]. Available: <https://webtorrent.io/faq>. [Accessed 7 3 2016].
- [50] T. Levent-Levi, "WebRTC P2P CDN: Where are the Use Cases?," BlogGeek.me, 9 3 2015. [Online]. Available: <https://bloggeek.me/webrtc-p2p-cdn-use-cases/>.
- [51] "Peer5 - P2P Video Streaming CDN FAQ," [Online]. Available: <https://www.peer5.com/faq>. [Accessed 8 3 2016].
- [52] "PeerCDN," 1 8 2015. [Online]. Available: <https://web.archive.org/web/>

- 20150810065820/https://peercdn.com/.
- [53] H. Andrioti, A. Stamoulias, K. Kapetanakis, S. Panagiotakis and A. G. Malamos, "Integrating WebRTC and X3DOM: bridging the gap between communications and graphics," in *20th International Conference on 3D Web Technology*, Heraklion, 2015.
  - [54] "XirSys," [Online]. Available: <https://xirsys.com/>. [Accessed 8 3 2016].
  - [55] "PeerJS - Simple peer-to-peer with WebRTC," [Online]. Available: <http://peerjs.com/>. [Accessed 8 3 2016].
  - [56] "OpenTok WebRTC Platform for Video, Voice and Messaging," TokBox, [Online]. Available: [www.tokbox.com](http://www.tokbox.com). [Accessed 8 3 2016].
  - [57] Mozilla Development Network, "MediaStream API," [Online]. Available: [https://developer.mozilla.org/en-US/docs/Web/API/Media\\_Streams\\_API](https://developer.mozilla.org/en-US/docs/Web/API/Media_Streams_API). [Accessed 1 3 2016].
  - [58] Mozilla Developer Network, "MediaDevices.getUserMedia()," [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia>. [Accessed 1 3 2016].
  - [59] W3C, "Media Capture and Streams W3C Editor's Draft 29 June 2015," Apple Computer, Inc., Mozilla Foundation and Opera Software ASA., [Online]. Available: <http://w3c.github.io/mediacapture-main/>. [Accessed 4 7 2015].
  - [60] Mozilla Developer Network, "RTCPeerConnection," [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection>. [Accessed 26 2 2015].
  - [61] W3C, "WebRTC 1.0: Real-time Communication Between Browsers," 15 2 2016. [Online]. Available: <http://w3c.github.io/webrtc-pc>.
  - [62] I. Grikorik, "Chapter 18. WebRTC," in *High Performance Browser Networking*, O'Reilly Media, Inc., 2013.
  - [63] S. Loreto and S. P. Romano, *Real-Time Communication with WebRTC*, Sebastopol, CA: O'Reilly Media Inc., 2014.
  - [64] A. Roach, J. Uberti and M. Thomson, "A Unified Plan for Using SDP with Large Numbers of Media Flows," 13 7 2013. [Online]. Available: <https://tools.ietf.org/html/draft-roach-mmusic-unified-plan-00>.
  - [65] J. Uberti, "Plan B: a proposal for signaling multiple media sources in WebRTC," Google, 3 5 2013. [Online]. Available: <https://tools.ietf.org/html/draft-uberti-rtcweb-plan-00>.
  - [66] N. Ohlmeier and B. Campen, "WebRTC in Firefox 38: Multistream and renegotiation," Mozilla.org, 25 3 2015. [Online]. Available: <https://hacks.mozilla.org/2015/03/webrtc-in-firefox-38-multistream-and-renegotiation/>.
  - [67] Monorail Chromium Open Issues, "Issue 465349: Need to implement WebRTC "Unified Plan" for multistream," 9 12 2015. [Online]. Available: <https://bugs.chromium.org/p/chromium/issues/detail?id=465349>.
  - [68] Mozilla Developer Network, "RTCDataChannel," [Online]. Available: <https://developer.mozilla.org/en/docs/Web/API/RTCDataChannel>. [Accessed 24 2 2016].
  - [69] W3C, "Screen Capture W3C Editor's Draft 03 July 2015," [Online]. Available: <http://w3c.github.io/mediacapture-screen-share/>. [Accessed 5 7 2015].
  - [70] T. Levent-Levi, "Screencasting as an extension – why is it any different than WebRTC

- video?," BlogGeek.Me, 1 9 2014. [Online]. Available: <https://bloggeek.me/screencasting-webrtc-different/>.
- [71] W3C, "Screen Capture W3C First Public Working Draft 10 February 2015," 10 2 2015. [Online]. Available: <https://www.w3.org/TR/screen-capture/>.
- [72] Y. Tian, Y.-C. Liu, A. Bhosale, L.-S. Huang, P. Tague and C. Jackson, "All Your Screens are Belong to Us: Attacks Exploiting the HTML5 Screen Sharing API," in *IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, 2014.
- [73] Mozilla Developer Network, "Same-origin policy," [Online]. Available: [https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin\\_policy](https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy). [Accessed 22 2 2016].
- [74] W3C, "Media Capture and Streams W3C Editor's Draft Section 10.1 NavigatorUserMedia Interface Extensions," [Online]. Available: <http://w3c.github.io/mediacapture-main/#navigatorusermedia-interface-extensions>. [Accessed 20 2 2016].
- [75] Mozilla Developer Network, "Navigator Interface," [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/Navigator>. [Accessed 20 2 2016].
- [76] Mozilla Developer Network, "MediaDevices Interface," [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices>. [Accessed 20 2 2016].
- [77] C. Alexandru, "Impact of WebRTC (P2P in the Browser)," *Internet Economic VIII*, pp. 39-58, 2014.
- [78] S. Dutton, "WebRTC in the real world: STUN, TURN and signaling," [Online]. Available: <http://www.html5rocks.com/en/tutorials/webrtc/infrastructure/>. [Accessed 20 2 2016].
- [79] J. Uberti and C. Jennings, "Javascript Session Establishment Protocol, General Design of JSEP," 25 2 2013. [Online]. Available: <http://tools.ietf.org/html/draft-ietf-rtcweb-jsep-03#section-1.1>.
- [80] M. Handley, V. Jacobson and C. Perkins, "SDP: Session Description Protocol," IETF, 6 2006. [Online]. Available: <https://tools.ietf.org/html/rfc4566>. [Accessed 3 3 2016].
- [81] J. Wright, "Session Description Protocol," Konnetic.
- [82] J. Rosenberg, R. Mahy, P. Matthews and D. Wing, "Session Traversal Utilities for NAT (STUN)," 10 2008. [Online]. Available: <https://tools.ietf.org/html/rfc5389>.
- [83] S. Dutton, "WebRTC in the real world: STUN, TURN and Signaling - HTML5 Rocks," 4 November 2013. [Online]. Available: <http://www.html5rocks.com/en/tutorials/webrtc/infrastructure>. [Accessed 10 September 2014].
- [84] S. Perreault, "NAT and Firewall Traversal with STUN / TURN / ICE," [Online]. Available: [www.viagenie.ca/publications/2008-08-cluecon-stun-turn-ice.pdf](http://www.viagenie.ca/publications/2008-08-cluecon-stun-turn-ice.pdf). [Accessed 5 3 2016].
- [85] Mozilla Developer Network, "WebRTC protocols," [Online]. Available: [https://developer.mozilla.org/en-US/docs/Web/API/WebRTC\\_API/Protocols](https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols). [Accessed 5 3 2016].
- [86] J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols," 4 2010. [Online]. Available: <https://tools.ietf.org/html/rfc5245>.
- [87] "STUN IP Address requests for WebRTC," [Online]. Available: <https://github.com/diafygi/webrtc-ips>. [Accessed 26 3 2016].

- [88] Mozilla Developer Network, "WebRTC connectivity," [Online]. Available: [https://developer.mozilla.org/en-US/docs/Web/API/WebRTC\\_API/Connectivity](https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity). [Accessed 5 3 2016].
- [89] J. Bankoski, P. Wilkins and Y. Xu, "Technical Overview of VP8, An open source video codec for the Web.," 2011.
- [90] A. Gal, "VP8 and H.264 to both become mandatory for WebRTC," 16 11 2014. [Online]. Available: <http://andreasgal.com/2014/11/16/vp8-and-h-264-to-both-become-mandatory-for-webrtc/>.
- [91] J. Ozer, "First Look: H.264 and VP8 Compared," 20 5 2010. [Online]. Available: <http://www.streamingmedia.com/articles/editorial/featured-articles/first-look-h.264-and-vp8-compared-67266.aspx>.
- [92] "Opus," [Online]. Available: <https://www.opus-codec.org/>. [Accessed 3 3 2016].
- [93] R. Screene, "WebRTC Audio Codecs: Opus and G.711," 5 12 2012. [Online]. Available: <https://thisisdrum.com/blog/2012/12/05/webrtc-audio-codecs-opus-and-g-711/>.
- [94] X. Marjou, S. Proust, K. Bogineni, R. Jesske, B. Feiten, L. Miao, E. Enrico and E. Berger, " WebRTC audio codecs for interoperability with legacy networks," 25 2 2013. [Online]. Available: <https://tools.ietf.org/html/draft-marjou-rtcweb-audio-codecs-for-interop-01>.
- [95] W3C, "HTML5 Canvas Element," [Online]. Available: <https://www.w3.org/TR/2011/WD-html5-20110525/the-canvas-element.html>. [Accessed 17 2 2016].
- [96] W3C, "HTML5 is a W3C Recommendation," [Online]. Available: <https://www.w3.org/blog/news/archives/4167>. [Accessed 17 2 2016].
- [97] W3C, "HTML5 Differences from HTML4 W3C Working Group Note 9 December 2014," 9 12 2014. [Online]. Available: <https://www.w3.org/TR/html5-diff/>.
- [98] P. Garaizar, M. Vadillo and D. López-de-Ipiña, "Benefits and Pitfalls of Using HTML5 APIs for Online Experiments and Simulations," in *2012 9th International Conference on Remote Engineering and Virtual Instrumentation (REV)*, Bilbao, 2012.
- [99] E. Bidelman, *Using the HTML5 Filesystem API*, O'Reilly Media, Inc., 2011.
- [100] W3C, "File API W3C Working Draft," 21 4 2015. [Online]. Available: <https://www.w3.org/TR/FileAPI/>.
- [101] W3C, "Media Capture from DOM Elements First Working Draft," 19 2 2015. [Online]. Available: <https://www.w3.org/TR/mediacapture-fromelement/>.
- [102] Mozilla Developer Network, "HTMLCanvasElement.captureStream()," 27 1 2016. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement/captureStream>.
- [103] Mozilla Developer Network, "HTMLMediaElement," 8 2 2016. [Online]. Available: <https://developer.mozilla.org/en/docs/Web/API/HTMLMediaElement>.
- [104] Google Groups, "discuss-webrtc Public Group," [Online]. Available: <https://groups.google.com/forum/#!msg/discuss-webrtc/fpbiC2HOgUY/ZTJN08NoGAAJ>. [Accessed 16 2 2016].
- [105] W3C, "MediaStream Recording Working Draft," 8 9 2015. [Online]. Available: <https://www.w3.org/TR/mediastream-recording/>.
- [106] Mozilla Developer Network, "MediaRecorder," 9 12 2015. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/MediaRecorder>.

developer.mozilla.org/en-US/docs/Web/API/MediaRecorder.

- [107] "jQuery," [Online]. Available: <http://jquery.com/>. [Accessed 15 2 2016].
- [108] B. S. Lerner, L. Elberty, J. Li and S. Krishnamurthi, "Combining Form and Function: Static Types for JQuery Programs," in *ECOOP 2013 – Object-Oriented Programming: 27th European Conference*, Montpellier, France, 2013.
- [109] B. Bibeault and Y. Katz, *jQuery in Action*, Manning Publications, 2008.
- [110] J. Resig, "Unobtrusive JavaScript," in *Pro JavaScript Techniques*, Apress, 2007, pp. 77-178.
- [111] W3Techs, "Usage statistics and market share of JQuery for websites," [Online]. Available: <http://w3techs.com/technologies/details/js-jquery/all/all>. [Accessed 15 2 2016].
- [112] T. Welch, "A Technique for High-Performance Data Compression," *Computer*, vol. 6, pp. 8-19, 1984.
- [113] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, California Technical Publishing, 1997.
- [114] "Node.js for the BeagleBone Black," ARMhf, 27 4 2013. [Online]. Available: <http://www.armhf.com/node-js-for-the-beaglebone-black/>.
- [115] W3C, "Media Capture from DOM Elements," 10 3 2016. [Online]. Available: <http://w3c.github.io/mediacapture-fromelement/>.
- [116] S. Dutton, "VP9 is now available in WebRTC," Google Developers, [Online]. Available: <https://developers.google.com/web/updates/2016/01/vp9-webrtc?hl=en>. [Accessed 18 4 2016].
- [117] C. Hart, "Does telephony matter if no one talks to each other anymore?," 18 1 2016. [Online]. Available: <https://medium.com/@chadwallacehart/does-telephony-matter-if-no-one-talks-to-each-other-anymore-5edf61f27e71#.gdmm244q7>.
- [118] P. Johnson-Lenz, "Rhythms, Boundaries and Containers: Creative Dynamics of Asynchronous Group Life," 1990.
- [119] E. Neely, "What is a Client Portal?," [Online]. Available: <https://clinked.com/2014/01/29/what-is-a-client-portal/>. [Accessed 5 7 2015].
- [120] D. Ristic, "WebRTC data channels for high performance data exchange," 4 2 2014. [Online]. Available: <http://www.html5rocks.com/en/tutorials/webrtc/datachannels/>.