Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Antony J.R. Meyn

Browser to Browser Media Streaming with HTML5

Master's Thesis
Espoo, June 30, 2012

Supervisors:	Professor Jukka K. Nurminen, Aalto University
	Professor Christian W. Probst, Technical University of Denmark

Aalto University
School of Science

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author:	Antony J.R. Meyn		
Title: Browser to Browser Media Streaming with HTML5			
Date:	June 30, 2012	**Pages:**	88
Professorship:	Data Communication Software	**Code:**	T-110
Supervisors:	Professor Jukka K. Nurminen Professor Christian W. Probst		

Video on demand services generate one of the largest portions of Internet traffic every day and their use is constantly increasing. Scaling up the infrastructure to meet this demand with the current model of Internet video delivery over HTTP, is proving to be very costly for service providers. An alternative model for video content delivery is the need of the hour to meet this challenge.

Peer-to-peer streaming is a viable alternative model that is highly scalable and can meet this increasing demand. The emerging HTML5 standard introduces APIs that give Web browsers the ability to communicate directly with each other in real-time. This also allows web browsers to behave as Peer-to-peer nodes.

In this thesis, we utilize these new APIs to develop a Video on demand service within the Web browser. The goal of this being, to determine the feasibility of such a solution and evaluate the usage of these APIs. We hope to aid the HTML standardization process with our findings.

Keywords: HTML5, P2P, WebRTC, Browser, Video on demand, Media, Streaming

Language: English

Acknowledgments

I thank Prof. Jukka K. Nurminen, from Aalto University, for his guidance and supervision in all phases of this thesis work. His knowledge and advice have been invaluable throughout the course of this work. I would also like to thank Prof. Christian W. Probst, Denmark Technical University, for co-supervising this thesis work.

I am grateful to Justin Uberti and Eric Bidelman from Google, for their guidance in helping me understand the various new HTML5 APIs and for their contributions via discussion and emails in furthering my understanding of the concepts involved. I would also like to thank Jouni Mäenpää from Ericsson Research Labs for introducing me to the related work that was being done at the labs, which was crucial during the early stages of this thesis work.

I thank all my friends who have provided their time and computational resources to help test the various modules from different geographical locations.

Last but not the least, I would like to thank my family for their constant support.

Espoo, June 30, 2012

Antony J.R. Meyn

Abbreviations and Acronyms

API	Application Programming Interface
DNS	Domain Name System
DOM	Document Object Model
DTLS	Datagram Transport Layer Security
FIFO	First In First Out
HTML	HyperText Markup Language
HTTP	HyperText Transport Protocol
HTTPS	HyperText Transfer Protocol Secure
ICE	Interactive Connectivity Establishment
IDE	Integrated development environment
IETF	Internet Engineering Task Force
IPv4	Internet Protocol version 4
JSEP	Javascript Session Establishment Protocol
JSON	JavaScript Object Notation
MIC	Message Integrity Code
NAT	Network Address Translator
P2P	Peer-to-Peer
PKI	Public Key Infrastructure
RFC	Request for Comments
ROAP	RTCWeb Offer/Answer Protocol
SCTP	Stream Control Transmission Protocol
SDP	Session Description Protocol
SGML	Standard Generalized Markup Language
SIP	Session Initiation Protocol
SMTP	Simple Mail Transfer Protocol
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
URL	Uniform Resource Locator

VOD	Video on Demand
W3C	World Wide Web Consortium
WHATWG	Web Hypertext Application Technology Working Group
WWW	World Wide Web
XML	Extensible Markup Language

Contents

Chapter 1

Introduction

The transmission of video and audio information is without doubt one of the greatest inventions the world has witnessed. The ability to see and hear events occurring far away, from the comfort of our homes was indeed a huge step forward. Thus, with the advent of what can be called the greatest communication network ever built, i.e. the Internet, it was only a matter of time before people were looking for ways to transmit and receive audio and video media over it.

The 'World Wide Web' (WWW)[7] or the 'Web' as it is more commonly known as, is the biggest part of the Internet, and a **web browser** is the software application that is used to retrieve, transmit and traverse information resources on the web. Web pages are designed using a language known as 'HyperText Markup Language' (HTML). Web pages or HTML documents are just plain-text documents containing HTML elements, which are tags enclosed in angle brackets within the web page content. These web pages reside on servers and are delivered to client devices (on request) via the 'HyperText Transport Protocol' (HTTP)[14]. As various vendors started implementing the web browser, HTML started to evolve as a language.

In its original form, HTML was only intended to build and display static web pages. However as the web evolved, browser vendors started looking for ways to make web pages more engaging and dynamic. As a step in this direction an interpreted client-side language now known as **JavaScript** was introduced. JavaScript quickly gained acceptance as a client-side scripting language for web pages and was soon adopted by all major browser vendors. **HTML5** is the latest version of HTML and apart from extending the markup for HTML documents, it also introduces 'application programming interfaces' (APIs) that can be used with JavaScript to design rich web applications.

The protocol used to deliver web pages from a web server to a web browser is HTTP, which is a **client/server** protocol. A web browser (client) initiates a request by establishing a 'Transmission Control Protocol' (TCP) connection to a web/HTTP server and requests a web page from the server. The server then responds by transmitting the contents of the requested web page over the TCP connection. The web browser uses the contents of the web page i.e. the HTML tags to interpret and render the contents of the web page on the clients browser.

In contrast to the centralized client/server networking model, there is also a decentralized **peer-to-peer** (P2P) networking model. "Peer-to-peer systems are distributed systems consisting of interconnected nodes able to self-organize into network topologies with the purpose of sharing resources such as content, CPU cycles, storage and bandwidth, capable of adapting to failures and accommodating transient populations of nodes while maintaining acceptable connectivity and performance, without requiring the inter-mediation or support of a global centralized server or authority."[47].

Simply put P2P systems are networks where the connected computers can act as both a client and a server, sharing their resources without the need for a central server. Each computer in a P2P network is referred to as a node. A P2P network requires that all nodes use the same or a compatible program to connect to each other on the network and share resources. It's interesting to note that the original vision for the WWW was closer to a P2P model, in that it assumed each user would actively edit and contribute to the content on the web, which is in contrast to the current broadcasting like structure of the web[6].

As mentioned previously, HTML5 introduces a number of new APIs that could change the way the Internet works. Until recently web browsers could only support the client/server network model, which meant that web browsers could only communicate with web servers. One of these new HTML5 APIs known as the **'Web Real-Time Communication' API** (WebRTC API)[3], introduces the capability of web browsers to communicate with other web browsers, i.e. it introduces P2P capabilities in the web browser.

The introduction of the capability of P2P communication or rather *Browser-to-browser* communication, has led to new use cases. The WebRTC API is currently being drafted by the 'World Wide Web Consortium' (W3C), but has already been implemented in the latest version of the major browsers such as Google Chrome, Firefox and Opera. Along with the WebRTC API there are several other newly introduced HTML5 APIs that facilitate the role of a web browser as a P2P media streaming node.

1.1 Research goals

Video on Demand (VOD) is a very popular online service as can be observed from the success of websites such as YouTube[1], Hulu[2] etc. The traditional way to implement these media streaming services has been to host the media files on a mini-cluster or a distributed system of servers. This helps maintain high-performance and high-availability of the media stream, but is a huge investment in hardware infrastructure for content storage and delivery. One possible way of reducing the bandwidth consumption and load on this back-end infrastructure would be to implement the streaming service with a WebRTC P2P streaming framework.

In this thesis, we look at an alternative method of streaming media on the Internet. Specifically a use case of Browser-to-browser streaming, and providing a VOD streaming service in the context of the new HTML5 WebRTC API. We attempt to implement a Browser-to-browser video streaming service using the newly introduced HTML5 APIs, to understand how a resource intensive application like media streaming can be designed to run completely within the web browser without the need for any native plugins. We also discuss the limitations encountered in this endeavor.

1.2 Problem statement

Consumer demand for streaming of video and audio on the Internet is continuously increasing. In the U.S. alone it is estimated that by 2015 movie downloads, IPTV and video streaming services could account for an annual total of several hundreds of exabytes.[51] An exabyte is 10 to the 18^{th} power (one quintillion) bytes or one million terabytes. In general, multimedia content has a large volume, so media storage and transmission costs are still significant. Bandwidth consumption by online video streaming services such as YouTube is only increasing. To ensure high availability, media streaming services host these audio and video files in clusters or distributed system of

[1]`http://www.youtube.com`
[2]`http://www.hulu.com`

servers.

Below are a few citations from white papers that scientifically predict the problems that would be created with video on the Internet, especially Video-on-Demand (VOD) services.
"It is estimated that in the United States by 2015, movie downloads and P2P file sharing could be 100 exabytes. Video calling and virtual windows could generate 400 exabytes. This roughly shows that in a period of 9 years the load on the Internet is expanding to about 50 times, and of course infrastructure would have to meet this growing demands, to avoid a crisis."[51]

Cisco systems have made their independent predictions in their white paper 'Entering the Zettabyte Era'[9]

- ***"Global Internet video traffic surpassed global peer-to-peer (P2P) traffic in 2010, and by 2012 Internet video will account for over 50 percent of consumer Internet traffic."*** *As anticipated, as of 2010 P2P traffic is no longer the largest Internet traffic type, for the first time in 10 years. Internet video was 40 percent of consumer Internet in 2010 and will reach 50 percent by year-end 2012.*
- ***"It would take over 5 years to watch the amount of video that will cross global IP networks every second in 2015."*** *Every second, 1 million minutes of video content will cross the network in 2015.*
- ***"Video-on-demand traffic will triple by 2015."*** *The amount of VOD traffic in 2015 will be equivalent to 3 billion DVDs per month.*

In view of the above predictions it is clear that if media streaming services continue to be delivered in the traditional way, the coming data flood will require increasingly huge investments in infrastructure to keep up with the demand. However as this is something we anticipate, we could definitely prepare for it. A closer look at VOD streaming service, allows us to make the following observations,

- It is a popular service via the browser.
- It is currently served over the client/server architecture, which makes it a very costly service from the server perspective.
- Distributing the load on a P2P topology would help reduce the cost on the server side.

Having identified a solution, in the next chapter we look into existing works that have tried to address this problem, and then conclude on how our solution would be different when implemented.

This thesis focuses on the feasibility of implementing a P2P based media streaming solution native to the browser using the available HTML5 APIs. We attempt to aid the community behind the standardization, by implementing and analyzing each module using only HTML5 related technologies.

1.3 Structure of the thesis

The rest of the thesis is organized as follows. Chapter 2 gives an overview of the related research and development in the field of P2P media streaming. Chapter 3 introduces HTML5 and then briefly talks about the various HTML5 APIs that are used in the implementation of our a P2P based media streaming solution. Chapter 4 gives a detailed description of the 'Web Real-Time Communication' (WebRTC) specification, and introduces the various working groups and APIs it covers. Chapter 5 describes the design of the various modules of our video streaming application, we then introduce the design of a video conferencing web application for the purpose of measuring the various experimental browser implementations of the new HTML5 APIs. Chapter 6 describes the HTML5 and JavaScript implementations of the various modules, and also evaluates their performance on the web browser. Chapter 7 discusses our findings and views of P2P with HTML5 from our experience of the implementation and evaluation. Finally in Chapter 8 we conclude the thesis and talk about the future work involved. The HTML5 and JavaScript source code of our implementations can be found in Appendix A.

Chapter 2

Related Work

In this chapter we look at previous work on media streaming solutions and solutions that have tried video streaming with P2P.

2.1 Video streaming

It is almost impossible today to find an Internet user who has not heard of the video sharing website YouTube[1]. YouTube popularized the concept of sharing user generated videos, which could be viewed by anyone with an Internet connection (before YouTube it was usually website owners who would provide the content). This impact of this concept is enormous. It offers a chance for anyone to become a celebrity overnight, almost literally. Eyewitness videos transform ordinary people into reporters, amateur movie makers have a platform where they can reach millions, and more importantly millions logon daily to be view these videos and be entertained. Even political parties have started using these sites in their campaigns[55]. The proliferation of affordable cameras, camera phones and availability of inexpensive tools for video editing and publishing also contribute to the success of the YouTube phenomenon.

Today YouTube is one of the most popular sites on the Internet. In fact as per the Alexa[2] traffic rankings, it is the third most viewed site globally. However hosting a site like this comes at a high cost. It was estimated that YouTube spends roughly $1 million a day just for server bandwidth in 2008[2]. This cost is only increasing daily, and is mainly due to the client/server architecture that it runs on. P2P delivery of videos has been considered as

[1]`http://www.youtube.com`
[2]`http://www.alexa.com`

an alternative[57] to alleviate this problem. We discuss existing P2P implementations that have tried to address this issue in the remainder of this chapter.

2.2 Peer-to-Peer streaming technology

In the client/server architecture a file is hosted by a server for download. The server has to serve every client download request by uploading the file within the constraints of its available upload bandwidth. Thus the available upload bandwidth is an upper limit in the design of the system, which becomes a bottleneck as the number of client requests increase. P2P file sharing protocols like BitTorrent are designed to overcome this problem.

Designing such a distributed system comes with a unique set of challenges, which include the unreliability of client infrastructure, participation, ensuring fairness to clients and the overhead of computation on the clients. But if a system is designed to overcome these challenges, there are a lot of benefits, such as quicker downloads and removal of bottlenecks on any central infrastructure.

2.2.1 BitTorrent

BitTorrent is a P2P file sharing protocol that can be used to reduce the server and network impact of distributing large files. It allows users to join a *swarm* of hosts where download and upload of the file happens simultaneously. The file is divided into small chunks known as *pieces*. A user who downloads a *piece* then becomes a source for upload of that *piece* to the other users. Thus the task of uploading the file is distributed among the clients who are also downloading the file[11]. BitTorrent is one of the most successful implementations of a P2P file sharing service, and as the BitTorrent protocol forms the underlying concept of our P2P Video streaming approach, it would help to understand its design.

When a file is to be hosted the owner of the file first goes through a publishing process using the BitTorrent client (a standalone software application). This process results in the creation of another small file which is called the "torrent" file. The torrent file contains meta-data of the file being shared, such as file length, name, hash for each *piece* and the *tracker* (the computer that co-ordinates the file distribution) Uniform Resource Locators (URL). The torrent file is usually then hosted on a regular web server for download.

Clients interested in downloading the file, download the respective torrent file and open it with their BitTorrent client. The BitTorrent client contacts the *tracker* to get a list of peers who have already downloaded the file completely or partially, and then connects to the various peers and requests the pieces of the file. Since the file is downloaded from many sources it is important to check the integrity of each downloading piece before merging it into the main file, the hash information for each piece is used for this this purpose.

BitTorrent redistributes the cost of upload to the downloaders, thus making hosting a file with a potentially unlimited number of downloaders affordable. It has been attempted by researchers before to find practical techniques for P2P file sharing before[8], however it was not previously deployed on a large scale because of the logistical and robustness problems involved. Figuring out which peers have what parts of the file and where the pieces should be sent is difficult to do without incurring a huge overhead. In addition, peers rarely connect for more than a few hours, and frequently for only a few minutes. Finally, there is the general problem of fairness. These issues are addressed by BitTorrent. A *tracker* addresses the problem of peers finding each other by providing a random list of peers (who are downloading the file) to a newly joined peer, and downloaders can make use of this to connect to each other. Peers maximize their download rates using a variant of *tit-for-tat*. To cooperate peers upload and to not cooperate they 'choke' peers ('choking' is a temporary refusal to upload). This also ensures fairness[10].

BitTorrent has also been found to be very effective in handling *flashcrowds*, a phenomenon in which a single file suddenly gains in popularity[35].

2.2.1.1 Streaming

BitTorrent at the basic level is focused on P2P file sharing rather than VOD streaming. It thus provide no guarantees about the order and timeliness of pieces to be downloaded. In VOD streaming however it is crucial that the pieces arrive in order and it time (before the media player has to display it). Also for online viewing of video (VOD), browsers are usually the choice of medium and requiring a separate software application or a plugin, which users would need to install for this purpose, hinders the user experience. We discuss these issues and some of the existing implementations of streaming solutions in the next section.

2.2.2 VOD solutions

To address the limitations of the basic BitTorrent protocol with regard to video streaming, several protocols have been introduced that extend the basic BitTorrent protocol. These solutions along with the implementations of these protocols are discussed here. They can be classified as 'Native applications' and 'Browser based solutions'.

2.2.2.1 Native applications

Tribler[36] is a BitTorrent based client application that has Video on Demand support. It is supported by the European Union 7th framework research program[3], with prior funding through research grants of the I-Share project, STW project and P2P-Next.

Tribler uses an overlay network for content searching, which makes it independent of external websites. The platform that Tribler uses was developed by P2P-Next and enables P2P based delivery of VOD and live streaming[30][29]. The *Give-to-Get* VOD algorithm used here discourages free-riding by rewarding peers which forward data to others. The peers that forward the most data will be provided with a better quality of service by their neighbours.The Give-to-Get algorithm also includes a piece-picking policy in which sets of pieces required for playback are divide into three subsets: high, medium and low priority. This allows a graceful transition between downloading pieces required on the short-term and those required on the long term with a distinct piece-picking policy within each priority set[28].

Spotify[24] is a streaming music service that uses P2P techniques. Data is streamed from both servers and a P2P network. One of the distinguishing features of the Spotify client is the low playback latency. It uses a proprietary client and protocol. Audio streams are encoded using 'Ogg Vorbis' with a default quality of q5, which has variable bitrate averaging roughly 160 kbps. The protocol is unsuitable for live broadcasts. A client cannot upload a track unless it has the whole track, this removes the overhead involved with communicating what parts of the track a client has. Clients use TCP between pairs of hosts and keep a connection open to the Spotify server. Using TCP simplifies the protocol design and implementation as TCP is reliable and includes congestion control.

Spotify uses two mechanism to locate peers. The first uses a tracker deployed

[3]`http://cordis.europa.eu/fp7/home_en.html`

in the Spotify back-end, this is similar to a BitTorrent tracker in functionality, the second mechanism uses a query in the overlay network. When a client asks for peers who have a track, the tracker replies with a maximum of 10 peers who are currently online. As clients keep a TCP connection open to a Spotify server, the tracker has a list of peers who are online at any given moment. Clients also send search requests to the overlay network, similar to the method used in Gnutella[41].

2.2.2.2 Browser based solutions

Most web browsers support the ability to add specific functionality via plugins or add-ons. One such plugin that implements P2P video streaming in a browser is the **SwarmPlugin**[4]. This is a plugin available for 'Firefox' and 'Internet Explorer' browsers, that is based on the VLC media player and the Tribler software (a European Union funded project). The SwarmPlugin has been put to use by the 'Wikimedia Foundation'[4].

Here the concept of P2P sharing brings in an additional advantage of distributing the content, rather than having it stored on a central server, thereby reducing the cost of back-end infrastructure. An overview of the system is presented in the Figure 2.1.

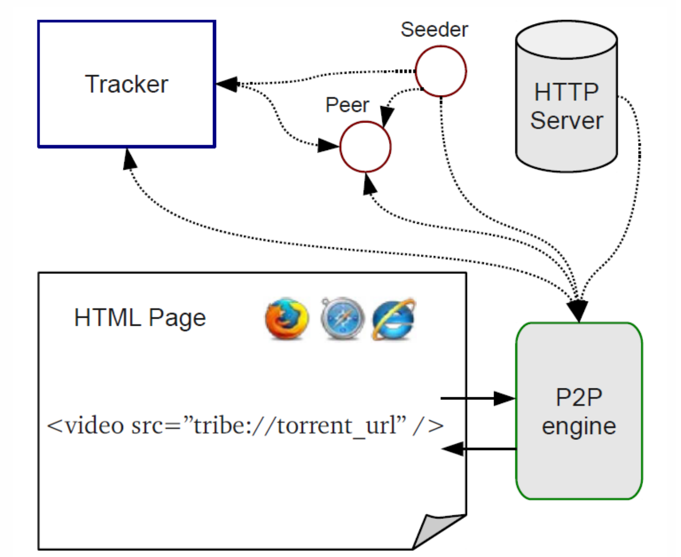


Figure 2.1: Architecture overview [4]

[4]http://trial.p2p-next.org/Beta

The design of this P2P solution addresses two key shortcomings of the earlier BitTorrent design with regards to VOD.

- Low delay in getting the initial few pieces.
- Placing the implementation within the browser.

Since the browser is the main target for VOD streaming, creating a plugin to implement this functionality is a step in the right direction. This plugin has been developed (initially only for the Mozilla browser) based on the P2P-Next[32] platform. The video player used is the HTML5 **<video>** tag, which uses the native implementation of video player within the browser. Although using a plugin is much more simpler than asking users to install a native application, the drawback of this is that the plugin needs to be implemented for every browser separately, and as a fallback the video would have to be streamed over HTTP in the traditional way. Effectively this means that users would need to install the plugin only to participate in the P2P functionality. A more desirable approach would be have this functionality implemented within the browser so that it is available by default.

With HTML5, browsers are making a concerted effort to be more standardized, and are attempting to move away from plugins, and towards building web applications that could be as powerful as native applications. If the functionality that was previously offered by a plugin becomes part of the functionality available in the browser itself, it is no longer considered a plugin. For example, Microsoft has decided to move to a plugin-free browser with the release of their 'Windows 8' operating system, and since Adobe Flash is a heavily used plugin they have decided to incorporate the implementation of Adobe Flash into the browser code base. In order to get cross browser support for native functionality, it is important to have a standard in place and this is the role of HTML5.

To conclude, it is quite clear now that one solution to reduce the network traffic load on VOD streaming servers would be to employ a P2P approach to distribute the network traffic among the nodes. However currently there exist no solutions that support this functionality fully within the web browser (which is the main medium used for viewing videos online) independent of third-party plugins (which as mentioned before require user intervention to install). In this thesis we attempt to design and implement an alternative solution inside a web browser using only currently available HTML5 APIs to achieve video streaming with P2P functionality.

Chapter 3

HTML5

In this chapter we look at the latest version of the 'HyperText Markup Language' (HTML) which is known as **HTML5**. This is the fifth revision of the HTML standard and is currently under development. HTML5 is intended to subsume the following three specifications,

- HTML 4.01 (HTML4)[19]
- eXtensible HTML 1.1 (XHTML1)[56]
- DOM Level 2 HTML (DOM2HTML)[16]

HTML5 also standardizes many of the features that have been used by web developers for years, but have not been formally documented by a standards committee. For instance the 'window' object which has been implemented by all web browsers has not been formally documented before the advent of HTML5.

HTML5 like its predecessors is platform independent as it is implemented within the web browser. The term HTML5 has been misunderstood, as it has sometimes been referred to as a platform and sometimes as a standard that is being designed and drafted. It is important to note that HTML5 is a collection of individual features, certain of these features are still in very early stages of being drafted. Therefore statements such as 'HTML5 supported browsers' are misleading, as most modern browsers support only a portion of these individual HTML5 features currently. As per the World Wide Web Consortium (W3C), in contrast to earlier HTML specification development models, the HTML5 specification will not be considered finished before there are at least two complete implementations of the specification. The goal of this change is to ensure that the specification is implementable, and usable by authors once it is finished.

HTML documents consists of elements and text, elements are usually represented as tags. HTML user agents namely web browsers, parse these documents to form the 'Document Object Model' (DOM). This is the internal data structure that represents a HTML document. The public interface of a DOM is specified in its 'application programming interface' (API). These DOM APIs can be used to inspect or modify the Web page dynamically within the browser by another programming language such as JavaScript. Unlike earlier specifications, the APIs and DOM are fundamental parts of the HTML5 specification. In addition to many new syntactical features such as a number of new tags, HTML5 also introduces several new APIs for development of complex web applications.

3.1 Evolution

HTML was initially considered to be an application of 'Standard Generalized Markup Language' (SGML) and was formally defined as such by the 'Internet Engineering Task Force' (IETF). The IETF is an open standards organization that develops and promotes Internet standards. The IETF closed its 'HTML Working Group' in 1996 after which the HTML specifications have been maintained (with inputs from commercial software vendors) by the 'World Wide Web Consortium' (W3C), which is the main standards organization for the World Wide Web.

In 2004, a majority of the W3C members voted against continuing work on HTML and decided to abandon it in favor of XML based technologies. In response to this a community of people interested in evolving HTML and related technologies, formed the 'Web Hypertext Application Technology Working Group' (WHATWG), which continued on the task of evolving HTML to what we now know as 'HTML5'. Later on in 2006, the W3C indicated interest in joining the HTML5 evolution and in 2007 started to work together with the WHATWG. The HTML5 specification was adopted as the starting point of the work of the new 'HTML Working Group' of the W3C.

Although the specification by and large is still being drafted and would take a few more years to complete ('Last Call' was in May 2011 and 2014 is the current target for 'Recommendation'), quite a few of the sections are stable. User agent browser vendors have started implementing most of these sections, and some of the vendors have even implemented certain sections which are not stable and still being drafted, for the purpose of providing feedback to the authors of the specification. It has been debated, as to whether the specification should be completed before implementation or vice-versa. It has been decided that they have to go along with each other, as completing the specification first, is quite likely to have implementation issues and the implementation feedback is important to modify the specification. And having the implementation done first, restricts the specification from being redesigned, as applications which rely on these implementations might stop working.[33]

3.2 HTML5 and JavaScript

JavaScript has been around much before HTML5 and is one of the most popular programming languages on the web today. It was originally designed as a lightweight interpreted client-side language that would appeal to non-professional programmers, and complement Sun Microsystem's 'Java' programming language. Because it runs at the client-side (usually a web browser), it can respond quickly to user actions making the application more responsive. The increasing speed of JavaScript engines has also added to the success of the language.

JavaScript is also one of the most misunderstood programming languages and was initially not taken seriously by many professional programmers. This misunderstanding is only exacerbated by its name. The prefix *Java* incorrectly suggests it is related to Java and the *Script* suffix suggests it is not a real programming language. Also the initial versions of the language did not have a lot of functionality such as error handling, inheritance and inner functions. Although these are supported now and the current version of the language is a complete object-oriented language, the previous versions created a low opinion about the language's capability. JavaScript was also known for some of its major design flaws, this had given room to a lot of bad programming, which added to the languages bad reputation and these bad designs cannot be easily removed, as applications depending on those flaws might fail. The literature on the language is also of very poor quality. There are in fact very few good books, which actually portray the language the right way.

Despite its flaws, at its core JavaScript is a very powerful, expressive programming language. It has also emerged as the only language that most popular web browsers share support for. It is the language of the web browser and now with HTML5 it is the only scripting language endorsed by the draft.

One of the main reasons JavaScript is popular as a browser language is because of its support for asynchronous execution of code. Although this feature slightly complicates the readability of the code, it brings in quite a few advantages which make up for this. For example from a user experience (UX) perspective this is important to ensure that web pages are responsive and take as little time to execute. The asynchronous features of JavaScript ensures this with function callbacks. A 'function callback' is a programming feature where methods may be passed to another method as an argument. This allows programmers to specify methods that can to be called on any event. This does require additional effort in understanding the code flow due to its non-linear execution path and hence a proper understanding of this programming paradigm is important for a developer to be able to write robust code. To better understand this let us look at the following code example, `setTimeout()` is an in-built JavaScript method that accepts two arguments,

```
setTimeout(function() { console.log('Hello'); }, 3000);
```

the first being a function callback and the second a duration in milliseconds. This method invokes the function specified in the first argument, after a time period equal to the duration specified in the second argument.

With the above code, the string 'Hello' is displayed after a delay of 3 seconds. If we now introduce another statement to display an additional string as shown below, we notice a difference in the output, This now prints the

```
setTimeout(function() { console.log('Hello'); }, 3000);
console.log(' World');
```

string ' World' first and then 'Hello' after a delay of 3 seconds. Thus the delay does not affect the code following it and execution proceeds. This example highlights the asynchronous nature of execution in JavaScript. This asynchronous or non-blocking feature of JavaScript is not to be confused with concurrency. Since the browser implementation decides how non-blocking code actually works, JavaScript is usually implemented as a single thread. Later on in this chapter we look at how multi-threading is implemented in JavaScript with the newly introduced HTML5 Web workers API.

In order to store and transmit information between web server and clients, we need to define a data interchange format. This brings us to another important in-built feature of JavaScript. JavaScript brings with it the **'JavaScript Object Notation'** (JSON)[12], which is a lightweight human-readable data interchange format. It is as the name suggests, derived from the JavaScript language. Despite this it is a language-independent format, with parsers available in most of the commonly used languages. The JSON format is used for serializing and transmitting data over a network connection, and is primarily used between web servers and clients replacing XML. A JSON representation is basically a collection of name-value pairs. The names and values are separated by the semi-colon character (:). An example of a JSON representation of an object that describes a person is shown below,

```
{
     "firstName": "John",
     "lastName" : "Doe",
     "age"      : 26,
     "address"  :
     {
         "streetAddress": "Baker Street, 5 C 391",
         "city"         : "Espoo",
         "country"      : "Finland",
         "postalCode"   : "02150"
     },
     "phoneNumber":
     [
         {
           "type"  : "home",
           "number": "358 456-7890"
         },
         {
           "type"  : "fax",
           "number": "358 123-4567"
         }
     ]
}
```

HTML5 consists of numerous individual features and in the remainder of this chapter we introduce those HTML5 features which are significant to our P2P VOD streaming implementation. These include the following,

- The <video> element, to allow native video playback within the HTML user agent.
- The File API, to access the file system.
- The IndexedDB API, to maintain and organize an index of the files and application settings.
- The Web workers API, that enables multi-threading.
- The Web sockets API, that allows bi-directional communication between the client and servers.
- The WebRTC API, that enables browser-to-browser communication.
- JavaScript helper libraries to parse the HTML DOM and to perform Message Integrity Checks.

One of the major challenges of dealing with HTML5 at this stage with regards to writing this thesis, is the immaturity of the implementations and changes in the API due to the ongoing standardization process.

3.3 The <video> tag

The **video** element is an important addition to HTML. It has been introduced for the purpose of playing videos and movies and to replace the <object> element, which was earlier used among other things to embed video content into web pages. The introduction of this element means that browsers would need to implement native video players so that videos can be played without the need for any plugins. Even though the video tag is a work in progress, it has been implemented by almost all major browsers to date. To demonstrate the media API and media events, the W3C has published a page [1], which when viewed with a browser that supports the <video> element, displays the properties, events and media types it supports.

When we talk about video players embedded in the browser, the next natural question would be which video formats are supported. This however is

[1] `http://www.w3.org/2010/05/video/mediaevents.html`

not clear at the moment as the current HTML5 draft does not specify which video formats browsers should support. Video formats are comprised of a combination of containers and codecs. Video containers are the file formats which contain the video stream and other related information, such as the meta-data of the video file, the title of the video, a thumbnail picture preview, the video stream bit rate, audio files with markers for synchronization and so on. Video codecs are the implementation of the algorithms used for rendering the video streams. Although it would be ideal if every browser supports a standard set of video formats, this is currently not the case. For this reason the video tag accepts multiple source URLs, and the browser plays the first resource that it supports. The online resource 'Dive Into HTML5'[2] details the browser support for the various video formats[33], and since this is constantly changing (and will continue to do so until standardized), it is important that developers are aware of this information.

3.4 File API

The HTML5 File API specification[37] is still being drafted by the W3C Web Applications (WebApps) Working Group. The implementation of this API within the browser would allow HTML5 applications to handle files via the browser in a standardized way. It has been designed to allow the web browser to create a sandboxed filesystem for each web domain, which means if we had two domains say *www.yahoo.com* and *www.google.com*, the files created by each of these domains would not be accessible to each other. The JavaScript code could request two types of filesystem storage, namely temporary and persistent storage. In temporary storage the files stored can be removed at the browser's discretion and in persistent it can only be removed by the application. Of course, neither of these would stop the user from manually removing these files from the hard disk if they so desired. The latter storage requires the user to grant access to the application.

It is important to note that the File API cannot be used to access the file system outside the browsers sandboxed filesystem. Providing such access needs to be debated and designed carefully, as it brings up security concerns. As depicted in the Figure 3.1, each browser that supports the HTML5 File API will have a separate sandboxed filesystem even if they are of the same domain. Files within each of these sandboxed filesystem cannot reference files that are outside their respective filesystem. This would mean that each

[2]`http://diveintohtml5.info/video.html`

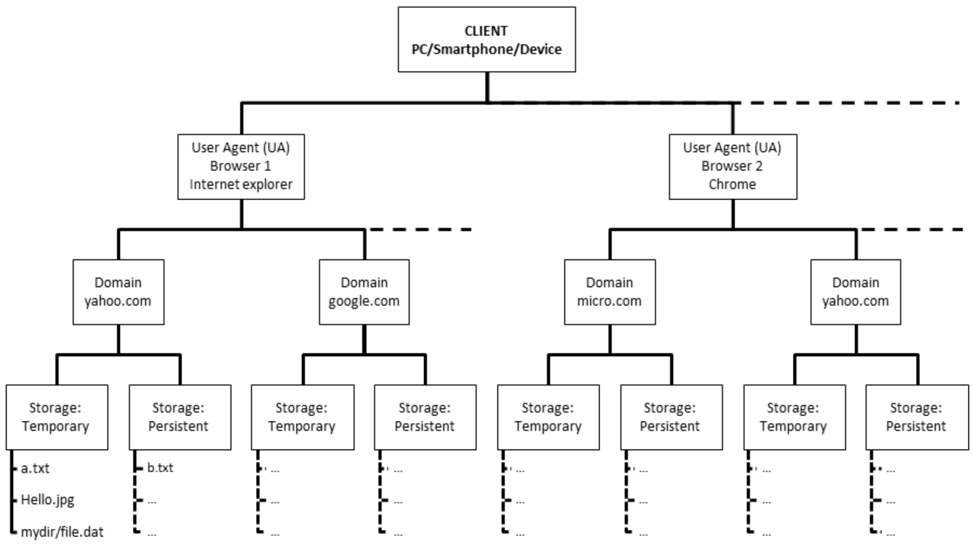


Figure 3.1: An overview of the HTML5 browser filesystem.

last node in the Figure 3.1 would not be able to reference files outside the respective node. This would not even be possible from the temporary to the persistent filesystem for the same domain within the same browser. Applications could be designed to allow the moving and copying of files within the temporary and persistent filesystem of the same domain and browser. In order to do this with different browsers, the domain web server would have to be used to relay the data.

The currently published version of the File API draft[**?**] has been implemented by major browser vendors.

3.5 IndexedDB API

The Indexed Database (IndexedDB) API[27] is another specification by the W3C Web Applications (WebApps) Working Group. The W3C group has recently been looking into the option of having a client side persistent database within the browser. This would give applications the ability to save information within the browser, so that they may access information even when offline. The WebSQL API was introduced so that information could be recorded and retrieved based on well established Structured Query Language (SQL) statements, however towards the end of the 2010 the W3C group decided that introducing SQL statements within JavaScript, was not a very compatible design and dropped it. This was then replaced by the IndexedDB API, which has a more JavaScript related API which deals with with data in the form of the JavaScript Object Notation (JSON). The HTML5 IndexedDB API allows applications to have a persistent storage of information locally within the browser. The availability of this stored offline information is the same as the File API discussed previously, where the information is saved separately for different domains and browsers.

The indexed database consists of object stores, these are the data structures that contain all the data. In relational databases these are known as tables. The indexed database consists of three types of transaction modes which allow access and modification to the object stores, they are 'readonly', 'readwrite' and 'versionchange' transactions. The 'readonly' transactions where data is not modified, can have many instances running in parallel, as these instances would not have any impact on each other. The 'readwrite' transaction on the other hand can only have one instance accessing a given object store running at a time, in order to ensure that transactions are mutually exclusive. It is also important to note that the object store schema cannot be modified in this transaction mode. Finally the 'versionchange' transaction mode is the only one in which the object store schema can be modified.

The IndexedDB API brings in a unique problem as to how the browser should handle the data when the object store schema has been updated. We should remember that when a new version has been deployed, all previous versions would be available with the user base, so if the designed application cannot afford to lose previously stored data, it would have to write code that handles the smooth transition of data from all the previous versions to the new object store schema. This is a problem that is specific to having a database on the client side, as server side databases would only have to be migrated from one schema version to another and would be reflected globally. This is also where the 'versionchange' transaction mode becomes important.

3.6 Web Workers

The Web Workers API[18] is another specification by the W3C Web Applications (WebApps) Working Group. One of the obstacles to porting of server-heavy applications to client-side JavaScript is the single-threaded environment of JavasScript. JavaScript does allowing asynchronous execution but this does not necessarily mean concurrency. As the asynchronous non-blocking code need not necessarily mean that threads are employed. In order to handle computationally intensive tasks, the HTML5 Web Workers API was introduced. This specifies an API to spawn background scripts in web applications.

With this it is now possible to run separate JavaScript files concurrently. Workers can utilize thread-like message passing ,to pass strings or JSON objects to JavaScript code that spawned the worker. Workers are meant for resource intensive task and are not meant to be used in large numbers as it could hog up the users system resources. It should ideally be used for CPU intensive scripts that would otherwise keep the application from being responsive.

It is important to free the worker by calling the close() method after it has completed its task. If not this might keep the allocated resources from being used. Workers are not thread safe, and for this reason they have some restrictions, such as accessing the DOM or the document object. Workers can spawn other workers, this can help break down resource intensive tasks to complete faster. Employing workers is an important design decision, which can help make heavy client-side computation quicker.

3.7 Web Sockets

The Web Sockets API[17] is also a specification maintained by the W3C Web Applications (WebApps) Working Group. A server push and a client pull are means of delivering content from a server computer (web server) to a client computer (browser). Unitl now browsers only supported client pull mechanisms, and server push was not directly provided to web developers. This is a major drawback, as servers were not able to inform clients when they had any updated information.

There have been several workarounds in achieving server push mechanisms they are namely Pushlets, Long polling, Browser plug-ins and third party

JavaScript libraries. Some of the best practices of abusing the HTTP protocol such as long polling have been discussed in RFC 6202[25]. The HTML5 WebSocket API is the first standardization to help browsers implement a common mechanism of server push delivery.

The WebSocket protocol[13] has been designed for the the WebSocket API, this protocol helps with bi-directional communication i.e. allowing a server to communication with a client and vice-versa. It helps to achieve this by using a single Transmission Control Protocol (TCP)[34] connection, rather than by creating a new TCP connection each time the server or client want to communicate with each other. This protocol is an independent TCP-based protocol, and is treated as a HTTP upgrade during the handshake procedure. Just like HTTP, it can work on the default port 80, and for secure connection it works on port 443 which is based on the Transport Layer Security (TLS) protocol[38].

Unlike most of the previously discussed HTML5 APIs, this Web Sockets API requires support from a server component as well. Hence it is not enough if the browser alone implements the Web Sockets API, the server would also have to implement the protocol to support this communication. There are a number of server side implementations of the Web Socket protocol in most of the well known languages like Python, C sharp, Java and even JavaScript. It might be a little surprising to know that JavaScript is used outside the browser, however this has been around for some time now (in fact it has been available soon after JavaScript was released for the browser in 1994). Node.js[3] is one recent notable example of a server-side implementation of JavaScript.

3.8 Additional libraries

In addition to the above discussed API's our HTML5 implementation of the Browser-to-browser video streaming application uses a few JavaScript libraries that are briefly described here.

3.8.1 jQuery

jQuery is a JavaScript library designed to simplify client-side scripting of HTML, It is designed to make it easy to navigate and manipulate the DOM,

[3]`http://nodejs.org/`

select DOM elements and handle events (among several others things). It simplifies web application development on the client-side and makes the JavaScript code more readable. We make use of this library to ease the development of our video streaming web application.

3.8.2 WebToolKit MD5

In our implementation of video streaming over a P2P network, we need to ensure the integrity of the pieces of the video stream received by a peer from another host on the network. One way to do this is with use of Message Integrity Codes (MIC). A MIC algorithm ensures that a given message will always produce the same MIC, assuming the same algorithm is used at both ends.

The MD5 Message-Digest Algorithm is a widely used cryptographic hash function that produces a 128-bit hash value[42]. MD5 is commonly used to check data integrity. An MD5 hash is typically expressed as a 32-character hexadecimal number. The WebToolKit MD5 script is a JavaScript implementation of the MD5 algorithm, and is used to process a variable length message into a fixed-length output of 128 bits. We make use of this JavaScript library to generate a MIC for each piece of the video stream received by the web browser.

The list of HTML5 APIs discussed in this chapter, is by no means exhaustive. HTML5 offers many more APIs for development of rich web applications. We have confined our discussion to the HTML5 features / APIs that are relevant to the implementation of our P2P VOD solution.

Chapter 4

WebRTC

In this chapter we take an in-depth look at the new HTML5 'Web Real Time Communication' (WebRTC) API that plays a key role in Browser-to-Browser communication. The WebRTC API introduces UDP based communication within the web browser, which is known for TCP based communication. This API has a separate W3C Working Group, known as the 'Web Real-Time Communications Working Group'. The API aims to allow Real-Time communication in Web browsers. Although the specification is in the stages of being a draft[5], experimentation is encouraged to help improve the specification. This thesis is part of one such experimentation, where we investigate how this API along with other HTML5 JavaScript APIs, can be used to build a P2P based VOD service, within the web browser.

Establishing Real-Time communication over the Internet is an existing concept, and proprietary software achieving this has been around for some time now. The WebRTC API aims to bring about the first specification to achieve inter-operability among Web browsers. To help achieve this, the specification is divided into two parts, the first being a protocol specification and the second a JavaScript API specification. The working group in charge of the protocol specification is the IETF Real-Time Communication Web working group (RTCWEB) and the working group in charge of the JavaScript API specification is the W3C WebRTC working group. The WebRTC working group continues the preliminary work done by the WHATWG on the PeerConnection API.

The API was first proposed by the WHATWG as the PeerConnection API, and it was later moved to the W3C when the WebRTC working group was formed in April 2011. The WebRTC working group had initially focused on a single use case, namely video conferencing, and for this reason the HTML5

Media Capture Task Force comprises of both the WebRTC Working Group in addition to the Device APIs Working Group.

The client-side technologies that currently fall under the charter of the WebRTC Working Group are as follows[1],

- To explore media device capabilities (camera, microphone, speakers).
- To capture media from the above devices.
- To process the media streams captured i.e. encoding and decoding.
- To establish direct peer-to-peer connections, including firewall/NAT traversal.
- To handle incoming media streams
- Delivery of the media streams to users.

The first three items shown above relate to the 'Stream API', and as VOD services do not involve live streaming from devices but rather streaming from static files, we only briefly describe the Stream API.

The 'Stream API' uses a MediaStream interface to represent streams of media data, such as the audio and video content from a camera device. Each MediaStream object can contain a number of audio or video tracks, with markers to enable synchronization when rendered by a user agent. A MediaStream object also has an input (for instance a local camera or a remote peer) and an output (for instance a HTML video element, a file or a remote peer). The JavaScript API to generate a MediaStream object from a camera or microphone is `getUserMedia()`.

4.1 Peer-to-peer connections

Until recently, Web browsers have been known as applications that only need to initiate outgoing connections. The introduction of the P2P paradigm to web browsers now introduces the need for them to become accessible over the network. In order for web browsers to have direct bi-directional communication with each other i.e. the ability to communicate without an intermediate server, we need to first understand how P2P connections are established over the Internet.

In order to identify a client on the Internet, an unique Internet Protocol (IP) address is allocated to it. The IP version 4 (IPv4) provides a little more than 4 billion such addresses, which is far less than the total number of connected devices to the Internet today. A popular tool for alleviating the consequence of IPv4 address exhaustion is 'Network Address Translation' (NAT)[48], which is a way to hide an entire address space behind a single IP address. This is achieved by modifying the IP address information in the IP packet headers while the packet is in transit across a traffic routing device (which performs the translation). The routing device performing this translation maintains a table to correctly handle the translation of the return packets. Using NAT a single IPv4 address can be shared by several clients behind the NAT while they are all connected to the Internet. However a NAT also makes it difficult for systems behind the NAT to accept incoming connections, which is a important step in P2P communication.

In a P2P network, nodes need to be able to connect directly to each other. However in some cases nodes may be located behind a NAT, which makes the establishment of a connection a little more tricky. There is no single guaranteed way to connect directly to peers in the presence of NAT's. 'NAT Traversal' is the general term used to describe the set of techniques that help peers create direct paths to each other in the presence of NAT's. To setup a connection peers usually make use of a signaling channel, which is used to exchange control messages that help setup the data session between the peers. Signaling channels are indirect connections between the two nodes, and can be used only to pass control messages between them (not data). The data session is then established using an offer/answer mechanism in the form of Session Description Protocol (SDP)[15] offers and answers[45]. We now discuss the NAT Traversal mechanisms in more detail.

4.1.1 NAT traversal mechanisms

NAT Traversal Mechanisms are required to create a direct communication path between nodes, if one or both of the nodes are behind a NAT. NAT's were implemented and used widely before they were standardized, this further complicates establishing connections through NAT's, as no single mechanism is guaranteed to work. For this reason, several NAT traversal mechanisms need to be used before a direct connection can be established. One such mechanism is **'Session Traversal Utilities for NAT' (STUN)**[44].

STUN is a standardized set of methods used by applications like real-time voice, video and messaging and other interactive IP communications. The

STUN protocol allows applications to obtain the public IP address and port number that the NAT has allocated for connections to remote hosts. This protocol requires assistance from a third party server (STUN server) located usually in the public Internet. The basic STUN protocol operates as follows. The client (inside a private network), sends a binding request to a STUN server. The STUN server sends a success response that contains the IP address and port as observed by it (this result is usually XOR mapped to avoid translation of the packets contents). Once a client has discovered its public IP address it can use this for communicating with peers by sharing this address, rather than the private address which is not reachable from peers on the public network.

Addresses obtained by STUN may not be usable by all peers. The addresses may or may not work depending on the network topology. Therefore STUN by itself cannot provide a complete NAT traversal solution. A complete solution requires a way by which clients can obtain a transport address from which it can receive data from any peer which can send packets to the public Internet (accomplished by relaying data through a server that resides on the public Internet). **'Traversal Using Relay NAT' (TURN)**[26] is a protocol that allows a client to obtain IP addresses and ports from such a relay. Although TURN almost always provide connectivity to a client, it is obviously at a high cost to the TURN server. Hence it is desirable to only use TURN as a last resort, preferring other mechanisms whenever possible.

To identify the optimal means of connectivity, the **'Interactive Connectivity Establishment' (ICE)**[43] methodology can be used. In the beginning of the ICE process both nodes are assumed to be ignorant about their network topologies, they might be behind a NAT or several NATs or even accessible directly. ICE allows the nodes to discover their topologies, thereby allowing them to discover one or more paths between them. In case there are no direct paths a TURN mechanism would always work as a last resort.

4.2 Specification and implementation

The WebRTC specification is going through several rounds of drafts and implementations within user agents. It was designed to have full control over the media plane, and to allow the JavaScript application to have control over the signaling plane. The reason behind this design was to allow the application developer to choose the protocol to be used for establishing the session, such as Session Initiation Protocol (SIP)[46] or the Jingle signaling protocol [50] or even an application specific protocol. In these protocols what gets sent across is the multimedia session description which contains all the necessary media and transport configuration details in order to establish the direct media connection between the two nodes.

The initial WebRTC P2P specification attempted to implement the signaling mechanism independent of the signaling protocol being used. This was done by exchanging SDP blobs via the signaling protocol. As a result of experimentation with this approach, a few disadvantages were observed. One shortcoming was that the user agent did not have sufficient context to determine the meaning of the SDP blob, i.e. to determine if the blob via the signaling channel was an offer or an answer or if it was a re-transmitted message.

The RTCWeb Offer/Answer Protocol (ROAP) specification[20] attempted to resolve the issues with additional structure in the messaging i.e. creating a generic signaling protocol to specify how the browser state machine should operate. ROAP messages are encoded in JSON, and it assumes that the signaling protocol would be implemented by the browser and the browser API would allow the application to request creation insertion of the messages into the state machine. ROAP was designed to be closely aligned with the PeerConnection API defined in the RTCWeb API specification.

Figure 4.1 illustrates a sample implementation that was implemented in Google Chromium web browser (other web browsers have also implemented the ROAP protocol for early experimentation of the specification), and is now referred as the `webkitDeprecatedPeerConnection()` API. We look into the reason about the name 'Deprecated' later. The Javascript API of this implementation worked as follows:

```
var PeerConnectionObj = webkitDeprecatedPeerConnection(
                                ICE_STUN_TURN_DETAILS,
                                function(sdp_object){});
```

The above method returns a PeerConnection object, and also has a function callback as the second parameter. The first parameter passed in is the ICE/STUN/TURN details. Once the user agent communicates with the STUN server, it receives the configuration details that need to be sent to the other user agent it intends to communicate with. An OFFER SDP blob

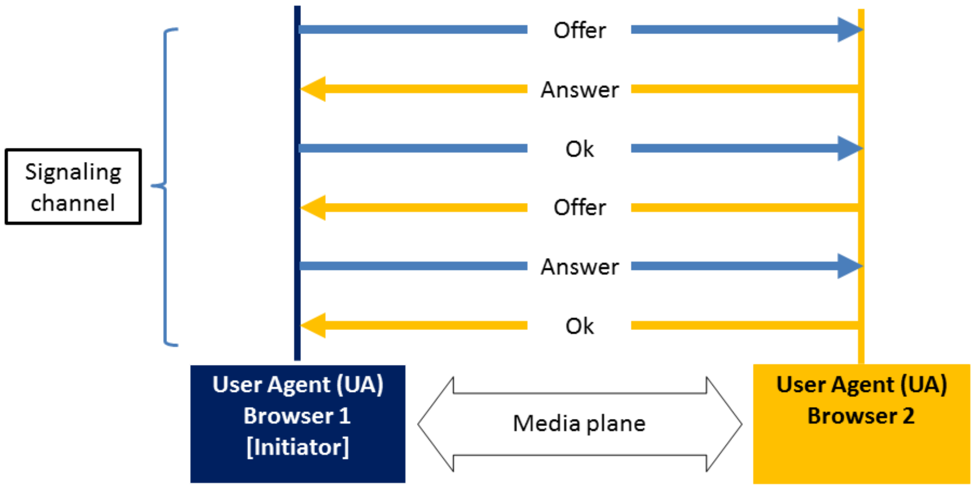

Signaling channel
Offer
Answer
Ok
Offer
Answer
Ok
User Agent (UA) Browser 1 [Initiator]
Media plane
User Agent (UA) Browser 2

Figure 4.1: WebRTC: RTCWeb Offer/Answer Protocol (ROAP)

is then created and passed to the function callback, the JavaScript application developer has to then pass this to the other user agent via the signaling channel.

```
PeerConnectionObj.processSignalingMessage(sdp_object);
```

The PeerConnection object has a method called the `processSignalingMessage()`, which takes the SDP object received via the signaling channel and after processing it, invokes the callback defined in the `webkitDeprecatedPeerConnection()` method, by generating and passing the next SDP message.

JavaScript Establishment Protocol (JSEP) [53]: Even though the ROAP protocol abstracts the signaling interactions, the state machine forces a least common denominator approach. For example, in the Jingle protocol the call initiator can provide additional ICE candidates after the initial offer has been sent, which allows the offer to be sent immediately for quicker call startup. However, in the browser state machine there is no notion of sending an updated offer before the initial offer has been responded to, making this impossible. The main reason this mechanism is inflexible is because it embeds a signaling state machine within the browser. Since the browser generates the session descriptions on its own, and fully controls the possible states and advancement of the signaling state machine, modification of the session descriptions or use of alternate state machines becomes difficult or impossible. To resolve these issues, the 'Javascript Session Establishment Protocol' (JSEP)[53] has been proposed. This pulls the signaling state machine out of the browser and into Javascript.

4.3 Data API

The Data API introduces the possibility of passing arbitrary data between browsers via a P2P connection, this is also an important feature to allow P2P VOD streaming, as it would allow streaming of the video directly from peer to peer. A general consensus has been reached at the RTCWeb working group to use 'Stream Control Transmission Protocol' (SCTP)[49] encapsulated on Datagram Transport Layer Security (DTLS)[40] protocol to send arbitrary data. *'The encapsulation of SCTP over DTLS over ICE/UDP provides a NAT traversal solution together with confidentiality, source authenticated, integrity protected transfers. This data transport service operates in parallel to the media transports, and all of them can eventually share a single transport-layer port number.'*[21].

This API has several interesting potential use cases such as, exchanging real-time game information like position and object state, file transfers between people while chatting, proxy-browsing (where a browser uses data channels of a peer to send and receive HTTP/HTTPS requests and data) etc.

The main reason for the long duration in drafting this specification are the security concerns involved, we discuss this in the chapters dealing with the evaluation. The specification of this API, is still in the very early stages of being drafted, and a completed implementation is expected within a year from now.

4.4 Standardization influence

This thesis started with an investigation on how we could implement a P2P based VOD service completely within the browser without any third-party plugins. The design and implementation of this have been quite challenging because most of the required functionality are still in the early stages of standardization and are changing constantly. Implementing modules to better understand the P2P paradigm of the browser has been like chasing a moving target. It has led to code that worked on one day, not working the very next day due to a change in the browsers code.

As an example, we initially used the `PeerConnection` API in Google Chrome, but this was deprecated and is now known as `DeprecatedPeerConnection`[1], which is again expected to change (or be dropped) as soon as JSEP is implemented in the browser.

[1] `http://www.webrtc.org/blog/peerconnectionisnowdeprecatedpeerconnection`

Chapter 5

Design

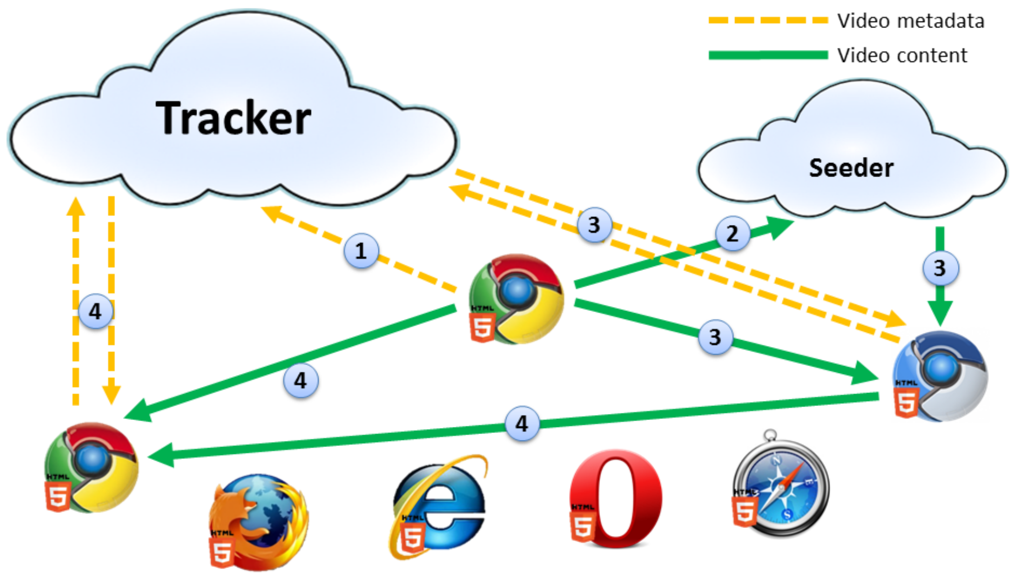
In this chapter, we discuss the design of the P2P VOD system that we attempt to implement. We make use of the HTML5 technologies discussed in the previous chapters, along with the knowledge gained from analysis of existing systems, to design a P2P VOD system within the browser that works with the WebRTC API. As this implementation is part of our thesis work with the goal of understanding the new HTML5 APIs and determining feasibility of a browser based solution, we focus on creating a robust application, that is quick and responsive with a minimum amount of processing overhead.

The design of our P2P VOD system can be divided into two distinct sections, namely the process of publishing videos and the process of their delivery or consumption. Since the features of HTML5 used are at various stages of implementation, we implement each of these sections individually and then evaluate the related aspects of the same. We found the most advanced HTML5 user agent (i.e. Web browser) to be provided by a company named Google, and have used its Web browser named 'Chrome' as the testing platform for our HTML5 implementations. We briefly introduce this Web browser in the section describing the implementation.

5.1 Network architecture

The three major entities involved in the design of our P2P VOD network are,

1. The Tracker
2. The Seeder (HTTP Server)
3. The Peers



1. Initially a peer creates a ‘tracker’ file containing the meta data of the video and the hashes of each piece of this video and uploads it to the tracker (server).
2. The actual video is uploaded to the “Seeder” (server).
3. When another peer tries to download this video, it would first obtain the respective tracker file and then start downloading the video pieces from the “Seeder” or the original peer or both
4. Eventually as the video gets popular, only the tracker would need to be in use, thus reducing the bandwidth utilization at the datacenter.

Figure 5.1: Network Architecture of our P2P VOD service

The Figure 5.1 shows the communication between these entities.

The Tracker is a server deployed on the cloud similar in functionality to a BitTorrent Tracker and keeps track of all the peers and their video content. It is used to store the video information i.e. the meta-data of each video file. These attributes are stored and delivered in the JSON format. The tracker supports bi-directional communication with each browser that is connected to it, and hence is also used as a signaling gateway to enable P2P communication between browsers.

The Seeder is a server, which stores all the videos provided by the VOD service. It is used to minimize start-up latency of playing videos by the client Web browsers (while P2P connections are being established), and also serves as a fallback mechanism to serve in video delivery. It is named the Seeder, and performs a role similar to an always available Seeder in the BitTorrent network. It would always be available to clients over the Internet i.e. on the cloud.

The Peers are the client Web browsers that playback the video. They load and execute the web application that facilitates the P2P VOD streaming. Each browser instance is considered a distinct peer, this means that two browsers running on the same device would also be considered as individual peers, as their file systems are not accessible to each other. Introducing a user authentication module to store user information on the cloud is beyond the scope of this implementation.

Having looked at the overview of out network, we move on to describe a more detailed design of the above mentioned entities in the context of the video publishing and consumption processes.

5.2 Video publishing

The video publishing process begins with a user starting the web application (i.e. browsing to a web page that contains our video publishing code) and selecting a video that they would like to share from a secondary storage device, such as their hard disk. We use a HTML input element to allow this selection, although this could be made more user-friendly with JavaScript code to allow drag-and-drop functionality, we have not implemented it as such in the interest of keeping our application simple and easy to understand.

Once a video file is selected, the web application has to process it for P2P consumption. This involves generating the meta-data of the video file for the tracker and moving the file to a location accessible to the web application i.e. the Web browser storage space. Below is the algorithm used to generate the meta-data of the video file,

Algorithm 1 Video file meta-data generation

```
Require: $videoFile$
  $numPieces := 0$
  $fileSize := getFileSize(videoFile)$
  $remainingBytes := fileSize$
  while $remainingBytes > 0$ do
      if $remainingBytes > 64kB$ then
          $pieceSize \leftarrow 64kB$
      else
          $pieceSize \leftarrow remainingBytes$
      end if
      $pieceFile = videoFile.read(videoFile.seek(numPieces * 64kB), pieceSize)$
      $mic \leftarrow computeMIC(pieceFile)$
      $micArray.push(mic)$
      $numPieces \leftarrow noPieces + 1$
      $remainingBytes \leftarrow remainingBytes - pieceSize$
  end while
  $fileName \leftarrow computeMIC(micArray)$
  $attributes.file\_mic\_array = micArray$
  $attributes.fileName = fileName$
  $attributes.fileMimeType = videoFile.codec$
  $attributes.fileDuration = videoFile.duration$
  $attributes.fileSize = fileSize$
```

As seen in the algorithm we use a piece size of 64 kB. This choice of size is based on two factors, the first being that it is an optimal file size to share pieces across a P2P overlay and the second is that the ‘Message Integrity Code’ (MIC) determination can complete in a reasonable amount of time for a file of size 64kB. It is important to ensure that the total time taken to determine the MIC for the pieces is minimum, as this is computationally intensive. The MIC serves a purpose similar to the hash of pieces in a Bit-Torrent file distribution. The file name by which the file is referred to in our P2P solution is the MIC of all the piece’s MIC’s. This ensures a unique file name across all the nodes in the P2P network, as long as the hashing algorithm used to determine the MIC is collision resistant.

5.2.1 Browser storage

The HTML5 File API plays a important part in the publishing process as it used for storage and distribution of the video file. The video publishing process requires storage space in the browser's file system to store the video file, as this is the only place from where the web application will be able to access it. Also the publishing device is a potential peer source for the same video. As mentioned in chapter 3, the HTML5 File API provides two types of filesystem storage, namely temporary and persistent. In temporary storage the files stored may be removed at the browser's discretion, but in persistent storage they are removed only when explicitly requested. Writing to the browser's persistent storage, requires consent from the user.

Initially the video file is written to the browser's temporary storage and processing is carried out from there, this is to avoid any conflict with existing videos files in the application's storage space (which is in the browser's persistent storage). It is not a problem if it overwrites another file with the same name in the temporary storage area. Once processing is completed and the video file meta-data is generated, the video file is ready for sharing. At this point the video file is written to the browser's persistent storage area and the meta-data is stored as a JSON object by the browser using the IndexedDB API. The use of the IndexedDB API allows faster querying/retrieval of the video information. The video is now available to our P2P web application for any further processing, without being affected by external factors, such as deletion or modification of the video file in the secondary storage device.

To make the video available to others the user chooses the publish option, at which point the computed meta-data is forwarded to the Tracker and the video file is uploaded to the 'Seeder'.

5.3 Video consumption

We now look at the design of the delivery or consumption system of the video files that are shared with our P2P solution. The main network entities involved in this are the 'Tracker' and the 'Seeder'. As mentioned previously the Tracker functions in a manner similar to a BitTorrent tracker, it is a Web server that contains the list of user published videos and the meta-data of those videos.

When a user launches our web application i.e. visits the Tracker (which is a Web server) and requests a video, the meta-data of the video file is downloaded to the user's browser along with a list of peers who currently have the video available (which would initially be 'Seeder' and the publisher). The web application then attempts to connect to each of the peers and start downloading the pieces of the video file from them. The MIC from the meta-data is used to verify each piece of the video file as it is downloaded. Once the entire video download is completed the meta-data (in JSON format) is saved in the web browser's database (IndexedDB) and the Tracker adds the details of the new peer to the list of available peers for that video. The video can now be played by the user's browser using the HTML5 <video> element.

All video files are first downloaded to the browser's temporary storage space, the reason we do not use the persistent storage space for this is because each time a video has to be stored by the application the user would be asked for permission. The user will be provided with an option for each viewed video to be available offline. If the user chooses to make it available offline then the video file would be transferred from the temporary to the persistent file storage. HTML5 also introduces the 'Offline Application Caching' API[54], which can be used to load and execute a copy of the HTML5 JavaScript code stored on the client, even when the web browser is not connected to the Internet. This would allow viewing of the videos that were made available offline by the user even when they are not connected to the Internet.

5.4 Media playback

Media Playback within the web browser is taken care of by the HTML5 <video> element, which can be used to render content from the browsers file system. As mentioned in the earlier chapter on HTML5, the video codecs and containers are currently not standardized, which means that each browser vendor can support a different set of video formats. This introduces two major problems for a P2P based VOD service, the first being, that a portion of the videos shared on the network, might not be possible to be rendered by certain user agents. Secondly, having different formats of the same video divides the peers based on all the available formats. For example if a video is uploaded in 2 different formats, the peers sharing videos in one format will not be able to share the video with web browsers that require it in the other available format.

We suggest converting all video files that are intended to be published to the most commonly available video format supported by leading browser vendors. This can also be done within the web application, since it would directly affect the video information details, and it also offloads the computation which would otherwise have be carried out on a central server. However this is beyond the scope of this thesis.

5.5 Peer-to-Peer communication

We introduced Browser-to-browser communication in chapter 4 on WebRTC. As mentioned previously, the HTML5 WebRTC API is fairly new and is still in the very early stages of implementation within leading browsers. The WebRTC Data API (of the WebRTC family of APIs) is what we require for our VOD P2P implementation, but as this API was only recently drafted, it has still not been implemented by any browser vendor as yet.

The WebRTC working group had initially focused on a single use case with the WebRTC API, which was video calls, for this reason the APIs related to this use case such as the WebRTC Stream API are fairly mature and have undergone several draft iterations. Some of these drafts have also been implemented for the sake of experimentation. In the WebRTC chapter, we also described how browsers can communicate with each other using the 'RTCWeb Offer/Answer Protocol' (ROAP). In the case of our P2P VOD solution, each browser would have to communicate with one or more browsers to get a single video file. This would mean that a browser would have to keep several peer connections open at the same time.

In the absence of the WebRTC Data API, to evaluate the browsers performance with several simultaneous peer connections open, we designed and implemented a P2P based video conferencing service using the WebRTC Stream API. This offers a close parallel to our use case and was the only way we could evaluate the browsers performance with multiple simultaneous peer connections.

As the Tracker has the list of currently active peers and supports bi-directional communication with each browser, we use it to setup the data sessions between the browsers by exchanging the Session Description details to create the peer connections between them.

The tracker has a unique connection ID to identify each web browser's connection to it. It has been implemented to receive three type of messages from web browsers that are connected to it, they are

1. **CREATE_ROOM_REQUEST**: The tracker generates a ROOM ID and responds to the web browser that sent this request.
2. **JOIN_ROOM_REQUEST**: The tracker adds the web browser's CONNECTION ID to the ROOM mentioned in the request, and responds with the list of remaining web browser CONNECTION ID(s) which are present in the room.
3. **B_TO_T**: (Browser To Tracker)This is the P2P Signalling Message sent from one web browser to another, the tracker redirects the message to the specified destination web browser.

The web application has been implemented to receive the following messages from the tracker,

1. **CREATE_ROOM_RESPONSE**: The web browser receives the ROOM ID and can share it with other web browsers that would like to join the video conference. This sharing is done manually.
2. **JOIN_ROOM_RESPONSE**: On receiving the list of web browsers in the room, the web browser creates a Peer Connection for each browser and starts sending P2P Signalling Messages to each of them via the tracker, in the form of B_TO_T described above.
3. **T_TO_B**: (Tracker To Browser) The web browser is expected to process these signalling messages sent by other web browsers via the tracker.

The Figure 5.2 shows the sequence of steps followed by three browsers in connecting to a conference hosted with our P2P based video conferencing service, It is important to note that a browser that intends to join a room, initiates the connections to all existing browsers in the room rather than the other way around. This is to avoid flooding the newly connected browser with connect requests, which it may not be able handle simultaneously.

Having seen the design of the various modules of our HTML5 P2P solution, in the next chapter we discuss the implementation and evaluation of each of these modules.

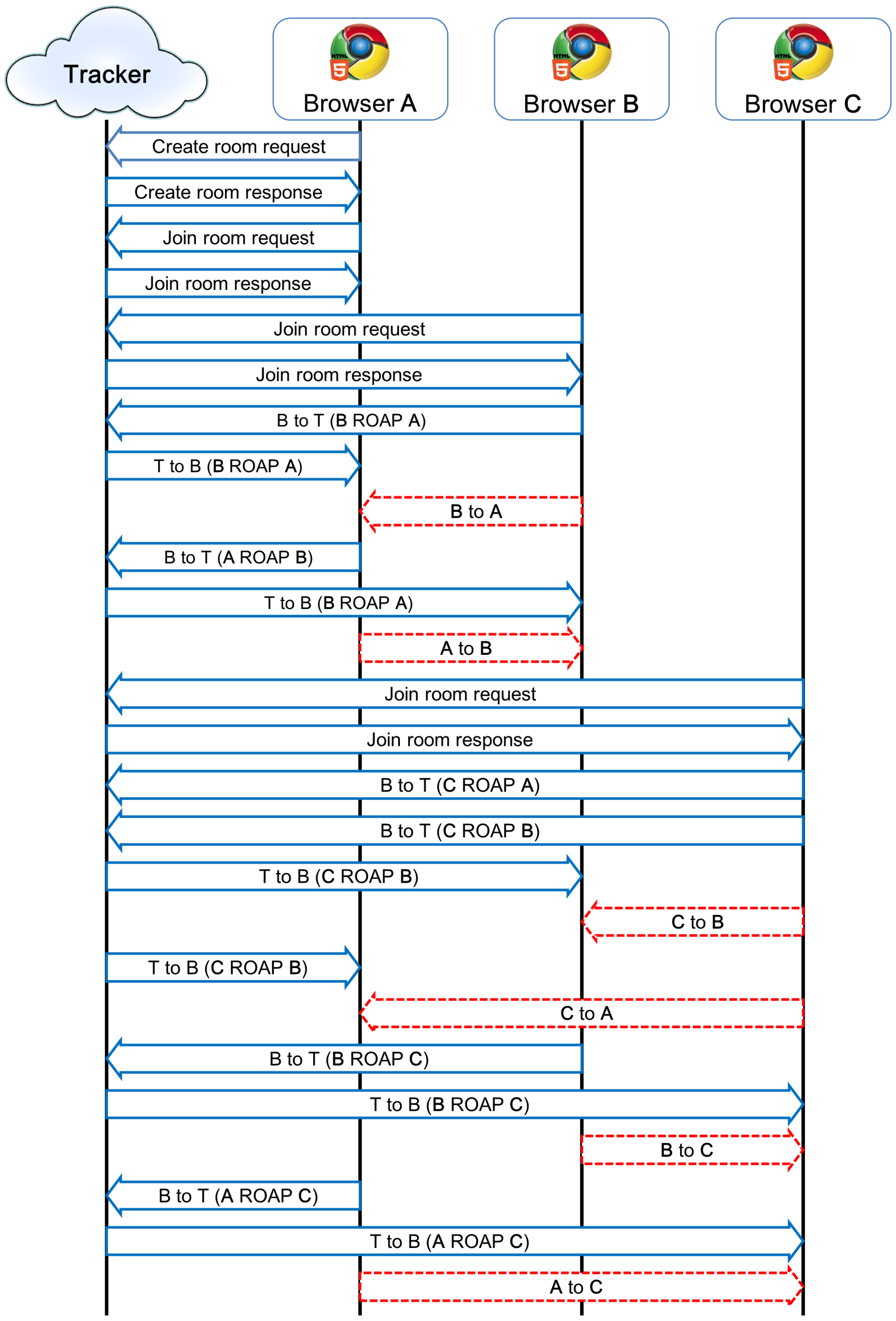


Figure 5.2: Message Sequence for Video Conferencing (Channel Mixing)

Chapter 6

Implementation & Evaluation

We now describe the implementation of our HTML5 P2P VOD streaming web application, based on the design discussed in the previous chapter. Our web application is written in JavaScript using the HTML5 APIs discussed in the earlier chapters. We also evaluate the performance of some of these modules that are implemented, and analyze the results. We also discuss some of the security concerns related to these modules.

Implementing a web application using HTML5 today is quite challenging, this has not so much to do with the technology itself but more to do with the fact that the HTML5 features, such as the WebRTC standard are not yet finalized. As we have seen, the HTML5 specification is still an evolving standard, and each of its feature and their associated APIs evolve independently. Since most of these features are still 'work in progress' drafts, the available Web browser implementations of these features are experimental. This means that our application's implementation is built on a constantly changing software layer, and this would be the case until all the individual features have a finalized specification and are implemented by all Web browsers.

Although HTML5 aims to bring about platform standardization for Web browsers, this is dependent on the fact that all Web browser vendors implement the feature specifications, this will however take a few more years, given the fact that most of the specifications are still being drafted and discussed by the individual working groups.

6.1 Development environment

Our choice of 'Development Environment' was the 'Visual Studio'[1] Integrated Development Environment (IDE) from Microsoft, as it eases the overall development process and also has a really good IntelliSense[2] implementation for JavaScript.

The Web browser we chose for the testing and evaluation of our HTML5 P2P VOD application, is from the browser vendor Google. Their user agent implementation was found to have the highest number of HTML5 features implemented at the inception of our thesis work. Google offers a public version of their browser implementation called "Chrome"[3]. Google also offers a 'Developer Preview' version of their user agent called "Canary"[4] and a third (and final) open-source version called "Chromium"[5]. We make use of all three versions, in the implementation and evaluation of each of the modules of our HTML5 P2P application.

6.2 Video publishing

We first describe the JavaScript implementation of the content publishing module's design. The data structure used to store the video information is shown in listing 1.

[1]`http://www.microsoft.com/visualstudio/`
[2]`http://msdn.microsoft.com/en-us/library/hcw1s69b(v=vs.71).aspx`
[3]`http://www.google.com/chrome/`
[4]`https://tools.google.com/dlpage/chromesxs/`
[5]`http://www.chromium.org/Home`

```
var movie_object = {
   "file_mic_array" : [],
   "slices": 0,
   "fileSize": 0,
   "fileName": "",
   "fileMimeType": "",
   "fileDuration": 0,
   "fileID": ""
}
```

Listing 1: Video file meta-data

Here the `file_mic_array` is an array to store the hash of each piece of the file. `slices` stores the number of pieces of the video file, this also includes the last piece which may be 64 kB or less. The remaining attributes are self explanatory. In order to slice the file, the HTML5 File API provides a `slice()` method that takes the start and end byte values as parameters. Since this API is not yet finalized, it is known by different names depending on the Web browser being used, for example it is currently known as `webkitSlice()` in Google's web browser implementations and `mozSlice()` in Mozilla FireFox, another Web browser implementation.

After slicing the file we have to calculate the Message Integrity Code (MIC) of each piece. Unfortunately HTML5 doesn't specify any JavaScript API for performing these computationally intensive task, which means that the browser doesn't perform these tasks natively. For this reason make use of the new HTML 'WebWorkers' API to spawn threads to calculate the MIC for each piece. We use the third-party JavaScript library 'WebToolKit MD5' to perform the MD5 calculation.

In order to communicate with threads the 'WebWorkers' API provides a message passing functionality, we use this to pass a JSON object containing the piece number and its content. On completion of each thread (i.e. computation of the MIC for a piece) the worker updates the `file_mic_array` with the computed MIC. After the MIC of every piece is computed, we compute the MIC of all the MIC's in the `file_mic_array` and rename the file with this resulting MIC. This name is used to uniquely identify the video file on the P2P network. A reference to this file is then added to the Web browser's database using the IndexedDB API. Adding the video information (meta-data) to the Web browser's database using the IndexedDB API is easy since it stores them in the form of JSON objects, so the `movie_object` data structure can be saved.

The code listing 2 shows the implementation of the WebWorkers thread that is spawned to compute the MIC.

```
var worker = new Worker('../scripts/md5/md5_check.js');
var input = {
  'slice': piece_pos,
  'plain_text': evt.target.result
};
worker.postMessage(input);
worker.addEventListener('message', function (e) {
  movie_object.file_md5_array[e.data.slice] = e.data.md5;
  if (md5_complete_counter === movie_object.slices) {
    renameFileAndAddToDb();
  }
}, false);
```

Listing 2: Invoking WebWorker md5_check.js

```
importScripts('../md5/webtoolkit.md5.js');

self.addEventListener('message', function(e) {
    var input = e.data;
    var md5 = MD5(input.plain_text, input.plain_text.length);
    var output = {
      'slice': input.slice,
      'md5':md5
    }
    self.postMessage(output);
    close();
}, false);
```

Listing 3: md5_check.js: MIC calculation of a piece

The code in listing 3, shows the MD5 (i.e. MIC) calculation for a piece of the video file (it returns a JSON object with the MIC and piece number, using message passing with the WebWorkers thread API). `MD5()` is a method from the third-party JavaScript 'WebToolKit MD5' library that we use for this computation.

These code snippets give an overview of how the module is implemented, the complete code can be found in appendix A.1 for a better understanding of the implementation.

6.3 Video consumption

We now look at the implementation of the video consumption i.e. the downloading and viewing of a video from within the browser. We have used multiple servers that host the same video file, since the Data API of WebRTC has not been implemented by any currently available web browser. First the video information object is downloaded from the tracker, and then the tracker provides a list of servers instead of peers that have the video file. The web browser then starts picking pieces of the file from the list of servers, in a round robin style, and in case any of the downloaded pieces do not match their respective MIC, the web browser attempts to download it from another server from the list. This implementation doesn't help from the

P2P perspective, but it helps understand how the web browser can handle the video file locally, in order to get a proper play back using the HTML5 video tag.

In this implementation it is important to note that although VOD streaming requires the pieces in sequence, JavaScript can only be used to request for each piece in sequence the call back mechanism doesn't guarantee that the requests are processed in a FIFO style, this brings in a challenge of assembling the file within the web browsers file system because it would be impossible to append a piece, unless all the pieces before it are first assembled. To overcome this problem, we first create a file with meaningless data, for the complete size of the original video file, it is then easy to seek the location of a piece after it has been verified and overwrite the dummy data with the valid piece data. It is important to monitor the download completion status of each piece, as VOD requires the initial few pieces of the video to have the valid video content. Similar to video publishing we use the WebWorkers API for performing the MIC verification and IndexedDB API to keep track of downloaded videos.

We let the video tag, play the video file from the browser's file system, only after a reasonable number of the initial pieces are downloaded and verified, to ensure that the video playback has a high probability of playing without interruption which is an important UX for VOD services. This reasonable number of pieces is dependent on the time required to download the complete video file. The time can only be predicted and not precise, as certain factors are dependent on external computing resources over the network. For our initial implementation we have taken the following into account, the latest known download bandwidth of the client(B), the size of a file piece(s), the total video file size(S) and video duration(d) which forms the video bit rate, we then calculate the approximate number of pieces that are need to be downloaded (N) with the following formula,

$$N = ((S/B) - d)B)/s \tag{6.1}$$

It is important to note that the **first N pieces** have to be downloaded, and not just N number of pieces, this is why we need to monitor the status of each piece in real time. The algorithm used to predict the time for download completion would differ in the case of a P2P network, as in that case, factors such as the number of peers sharing the video file is an important factor to determine the the time required to download the file.

Having looked at the basic implementation of video publishing and consumption, we now perform some measurements on these implementations and try to analyze the results.

6.4 Peer-to-Peer communication

As mentioned in chapter 5.5, the HTML5 'WebRTC Data' API which is required for the actual P2P data transfer of files, is not implemented by any Web browser vendor as yet. Hence to evaluate Browser-to-browser communication we implemented a video conferencing application, using the WebRTC Stream API (which is implemented in Google Chrome). We use two HTML5 APIs, the WebSocket API and the WebRTC API. The bi-directional communication between the Tracker and the browsers is implemented using the WebSocket API, and the communication between web browsers is implemented to be P2P using the WebRTC API.

As explained earlier in the design the current implementations within the browser are based on ROAP, but in order to implement a video conferencing application, the signaling server which in our case is the tracker has to ensure that the web browsers exchange Session Description Protocol messages correctly, this process is know as mixing i.e. multi-channel support. We have used 'node.js' to implement the server side signaling that uses a WebSocket JavaScript implementation. The messages that are passed between the signaling server and Web browser are JSON objects, which does not require any special formatting as both the server and client are implemented using JavaScript.

The Tracker generates a unique connection ID to identify each web browser's connection to it. Its implementation allows it to receive and handle the three types of messages (CREATE_ROOM_REQUEST, JOIN_ROOM_REQUEST and B_TO_T) from web browsers that connect to it. In the case of our VOD application the Tracker would have to pool web browsers based on video files being shared, just like it is now pooling them based on rooms.

6.5 Evaluation

6.5.1 Integrity check module

In this section we measure the JavaScript implementation of MD5 hashing, that is used extensively for piece verification in our VOD design. We compare it to a native implementation and measure the difference in time taken by each of these solutions.

File Size	JavaScript	Native
15 MB	7 seconds	0.3 seconds
30 MB	26 seconds	1.3 seconds

Table 6.1: MD5 computation time in JavaScript vs Native implementation

As can be surmised from the above, the native implementation of the MD5 hashing algorithm performs much faster than its JavaScript counterpart. In chapter 7 we propose an addition to HTML5 that could help address this computationally intensive task.

6.5.2 File storage limitations

Although the HTML5 specification does not limit the size of the file that can be handled within the web browser's file system. In our implementation we identified that the web browser crashes unpredictably as the file size increases. This means that we cannot precisely measure the file storage limitations. We have been able to process video files for upto 60 MB after which the browsers behavior becomes unpredictable (In times we have beeen able to process files of larger sizes). One way of working around this would be to redesign the processing of the file as smaller chunks. As these APIs are not in the stable release of the browser we have reported a bug[6] to the browser vendor for further investigations, to better understand a solution.

6.5.3 Multiple peer connections evaluation

In this section we evaluate the WebRTC Stream API, using the implementation of video conferencing application. Since video conferencing enables continuous video streams to be exchanged between browsers, they would generate significant network traffic and CPU utilization. We measure these values for various scenarios, on three different devices.

[6]Issue 94589 of the Google Chromium Browser, URLhttp://crbug.com

Since all these video streams are continuous, we can assume that they utilize the maximum network capacity of each Peer Connection object.

Scenario 1: We first measure the rise in CPU utilization and network traffic, when the browser on the device starts video conferencing with another browser. This means that each browser would have two connections open per peer, one for the incoming video stream and another for the outgoing video stream.

M1 : PC running the Windows 8 operating system.
M2 : Mac operating system.
M3 : Laptop with Windows 7 operating system.

Device	Increase in CPU utilization	Increase in network traffic
M1	9%	4.1 Mbps
M2	9.19%	Not Available
M3	15%	5 Mbps

Table 6.2: Measurements of resource utilization in Scenario 1

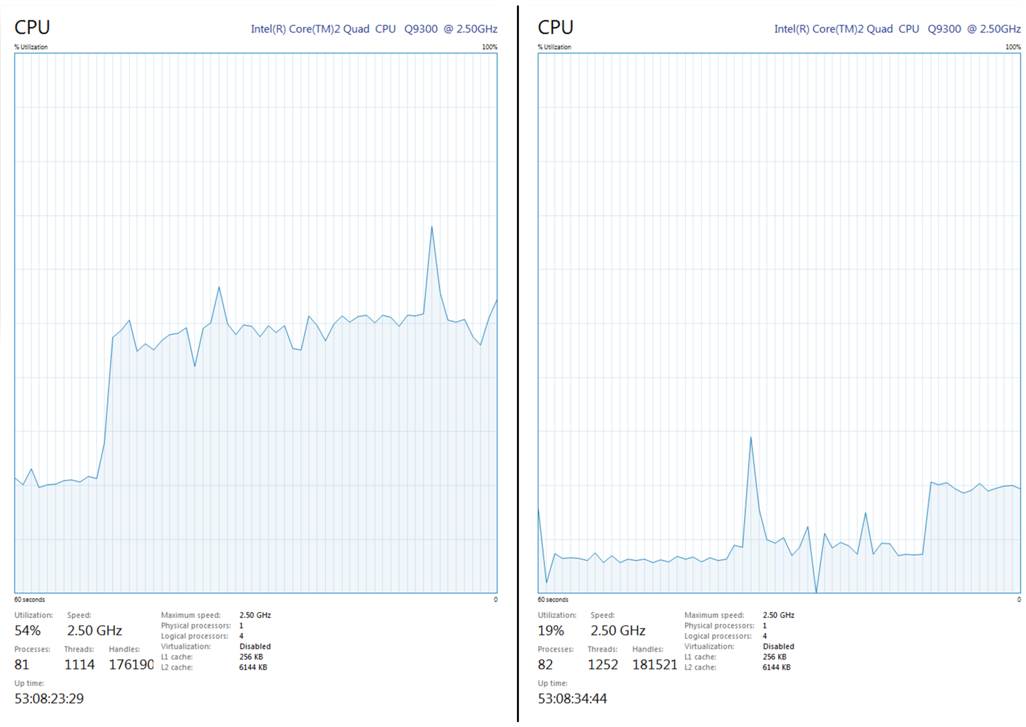
Scenario 2: In this scenario we similarly measure the device performance from switching from the previous scenario to accommodate another web browser. This would mean that each browser would have four peer connections i.e. two incoming video streams and two outgoing video streams.

Device	Increase in CPU utilization	Increase in network traffic
M1	35%	2.2 Mbps
M2	4%	Not Available
M3	22%	3.4 Mbps

Table 6.3: Measurements of resource utilization in Scenario 2

Figure 6.1 shows the CPU utilization of Scenario 2 on the left and Scenario 1 on the right, for the device M1.

The results clearly indicate that each peer connection incurs significant CPU utilization when used for continuous streaming. This is a concern for the use case of P2P Video Conferencing, as the application would require several simultaneously open peer connections. In the case of P2P VOD streaming we would need to ensure that the CPU utilization of the WebRTC Data API would be reasonably close to its client/server counterpart.


CPU
Intel(R) Core(TM)2 Quad CPU Q9300 @ 2.50GHz
% Utilization
100%
60 seconds
0
CPU
Intel(R) Core(TM)2 Quad CPU Q9300 @ 2.50GHz
% Utilization
100%
60 seconds
0

Utilization: 54%
Speed: 2.50 GHz
Processes: 81
Threads: 1114
Handles: 176190
Up time: 53:08:23:29
Maximum speed: 2.50 GHz
Physical processors: 1
Logical processors: 4
Virtualization: Disabled
L1 cache: 256 KB
L2 cache: 6144 KB

Utilization: 19%
Speed: 2.50 GHz
Processes: 82
Threads: 1252
Handles: 181521
Up time: 53:08:34:44
Maximum speed: 2.50 GHz
Physical processors: 1
Logical processors: 4
Virtualization: Disabled
L1 cache: 256 KB
L2 cache: 6144 KB

Figure 6.1: M1: Scenario 2 and Scenario 1, CPU utilization

Chapter 7

Discussions

In this chapter we discuss a few topics in relation to our implementation experience and results.

7.1 Homogeneous & heterogeneous P2P

The design and implementation of our proposed VOD system would be able to share video content over a homogeneous P2P network, since web browsers would be able to communicate with only other web browser running the same HTML5 and JavaScript code. In other words, they cannot share content with existing P2P networks such as BitTorrent and vice versa. A heterogeneous implementation would naturally be desirable to gain access to the huge archives of video content that is already available on well established P2P networks, such as implementing a BitTorrent client with JavaScript that runs within the web browser, but since P2P has only recently been introduced to web browsers, we found it easier to aim for a homogeneous design and implementation.

In this section we discuss the reasons a heterogeneous P2P implementation is not yet quite possible. We have already talked about HTML5 and JavaScript bringing platform in-dependency to applications, as they can be developed once and then work on any modern web browser that conforms to the standards. This makes the web browser move into the direction of being a virtual machine. The role of the HTML5 specifications of the various features, limits the capability of such a platform, for good reasons such as steady browser inter-operability and security concerns. Let us take an example of a BitTorrent implementation within the browser. In order to implement such a HTML5 client, the protocol would have to be first be implemented within the web browser before which a working group would have to create a specification of the protocol, specific to the web browser. Only after this would a JavaScript API to this interface allow it to be possible to implement an application that uses the BitTorrent client. It should clearly be noted that this is not a limitation of the JavaScript language, as this language could still be used to implement a BitTorrent client to work on ‘node.js’, which is server side JavaScript environment.

An alternate way of getting access to the video content would be to have a heterogeneous P2P client running on the always on ‘Seeder’ of our existing design, this would help provide access to the huge archives of video content, but not the benefits of P2P to the web browser.

7.2 Mobile platforms

Mobile platforms have gained huge popularity with the advent of ‘smartphones’, tablets and laptops. These devices come in various hardware denominations and software platforms. They are a testimony to Moore’s law on the computer hardware, written in 1965 ’whereby the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years’[31].

Several multinational companies such as Microsoft, Apple, Google are competing to build more of these powerful mobile hardware and software platforms for the general public. This is leading to a huge network of heterogeneous mobile devices in the market. Although these mobile platforms are of different operating systems, one thing in common they all have is the ‘Web browser’. This makes applications built using HTML5 and JavaScript applications and ideal choice for these platforms. Eventually when HTML5 is standardized and all these Web browsers conform to it, web applications would work on all of these platforms.

One of our initial goals was to build a VOD P2P service for mobile devices such as smartphones, but after more research building it with HTML5 appeared to be a better choice, as the service would eventually work on the smartphones and mobile platforms too. We have also managed to tested parts of our implementation on Microsoft’s ‘Windows 8’ tablet operating system that can run Google’s Chrome Web browser.

7.3 Security concerns

Another important aspect of HTML5 is security, as Web browsers are a constant target for attackers who are always attempting to benefit from every single vulnerability of the Web browser. Security is a main concern for the W3C working groups as well. The HTML5 community is very responsive on this front and would like to ensure that security concerns are not left entirely to the web browser vendors. Also, HTML5 applications have more restricted access to system resources than with Flash. We look at few of the major security concerns related to the HTML5 APIs that we used in our application.

With the advent of the networking APIs in HTML5 such as WebSockets and WebRTC, one possible attack is the cross protocol attack. 'These are attacks in which the attacking script generates traffic which is acceptable to some non-Web protocol state machine.'cite For example If a user has access to an Simple Mail Transfer Protocol(SMTP)[22, 23] server to send emails. An attacker could gain access to this restricted service by hosting a web page with JavaScript code to communicate with the SMTP server and send an email. When legitimate user of the SMTP server opens this web page on a web browser, the script could perform the attack. A more detailed illustration of this attack has been described in the paper 'The HTML Form Protocol Attack'[52]. 'In order to resist this form of attack, WebSockets incorporates a masking technique intended to randomize the bits on the wire, thus making it more difficult to generate traffic which resembles a given protocol.' This security concern is also applicable to the WebRTC Data API, which is still in its early stages of being drafted[39].

Another attack is the Cross Directory Attack, in this attack web applications gain access to the sandboxed HTML5 file system of another web application. Web browsers are built to create a sandboxed file system for each web domain. This means that a JavaScript application hosted on www.abc.com cannot access the filesystem of another application hosted on www.xyz.com. There are two simple ways this security mechanism can be compromised, one method would be to spoof the Domain Name Service (DNS) of a website, this way the browser would give the attacker's application access to the victim's file system. Another way is not really an attack, but rather an interesting loop hole in the security mechanism. If both the attacker and victim have web applications hosted on the same domain. For example, on a free web hosting service, such as 'DropBox'[1] (which allows users to host pages of their

[1] `www.dropbox.com`

own in public folders), since the domain for the applications is the same (i.e. DropBox in this case), they would share a common file system on a client's web browser.

7.4 Security API proposal

HTML5 is the first step forward, to building powerful web applications to match native implementations. As we have clearly understood HTML5 is collection of many individual features. In the previous chapter we compared the performance of a MIC algorithm, implemented in JavaScript to a native implementation , the results clearly indicate that the native implementation performs much faster for this CPU intensive piece of code. The question now arises as to how we could achieve the same level of speed and efficiency in a web application.

Most (if not all) modern Web browsers, contain implementations of the various security algorithms that provide functionality such as hashing, encryption and decryption. These libraries are present within the we browser in order to support protocols such as HTTPS, certificate verification in the 'Public Key Infrastructure' (PKI) etc. However they are not accessible to web developers for use in their applications via JavaScript. We would like to propose that these APIs be made available to web developers, along with a standard set of security algorithms to avoid having to implement these security algorithms in JavaScript. This would be an great addition to the features that will be made available with the HTML5 APIs.

Chapter 8

Conclusion

We investigated the use case of a VOD service over a P2P network topology within the web browser, using only HTML5 which comes along with libraries of individual features as JavaScript APIs. The design required a clear understanding of the WebWorkers, WebSockets and WebRTC APIs. We then designed, implemented and evaluated the modules and used this information to investigate alternatives to improve the performance of modules that required processor intensive tasks. We then looked into the new feature of Broswer-to-browser communication using the WebRTC API. The WebRTC API introduces UDP based communication within the web browser.

As the specific API (WebRTC Data API) within the WebRTC is in the early stages of being drafted and not yet implemented (even experimentally) by any of the Web browser vendors to date, we used an alternate use case (video conferencing application) to design and evaluate the simultaneous peer connections within the web browser.

Unlike native applications, currently building a HTML5 based solution is quite challenging, given the ongoing standardization process with changing specifications and experimental browser implementations. This thesis has focused on the feasibility of implementing a P2P based media streaming solution native to the browser using the available HTML5 APIs. We attempt to aid the community behind the standardization i.e. the working groups, by implementing and analyzing each module using only HTML5 related technologies. We realize that building a complete browser based solution of a P2P VOD solution requires some more time, as the working groups and browser vendors work on designing and implementing the HTML5 standard. We also discuss of the HTML5 specific issues in browser-to-browser media streaming.

8.1 Future work

It is quite evident, that building a P2P based VOD service using only the HTML5 standard has a longer roadmap, as it involves several stake holders, which includes working groups and web browser vendors. In order for one to proceed further in implementing a complete working solution, it would be important to join the related working groups, and help shape the future of the Internet usage by participating in the discussions that help draft the specifications. These discussions are officially done through mailing groups that are publicly archived.

JavaScript was initially designed as a client-side language that would appeal to non-professional programmers, this has led to a huge group of programmers, who would find the new HTML5 JavaScript capabilities quite hard to understand and use the right way. For this reason the more talented group of programmers create open source third party JavaScript libraries, that abstracts the core implementation and gives an easy to use interface for the application developer. The HTML5 APIs such as the IndexedDB API and WebRTC API provide a good case for building such libraries, so that they can be used by application developers easily. For example, in the WebRTC API it would be possible to expose it as a third party JavaScript library, where the application developer does not need to get into the complexities of NAT traversal.

Efficient algorithms for the tracker have to be designed specific to the HTML5 P2P VOD service. This work does not end by implementing a P2P based solution for media streaming, measuring and analyzing the performance is important to see how it would work on all the various platforms. The participation of mobile platforms would be an interesting research area, once there are stable HTML5 supported Web browsers for these platforms.

Bibliography

[1] Web Real-Time Communications Working Group Charter. http://www.w3.org/2011/04/webrtc-charter.html.

[2] YouTube looks for the money clip. http://tech.fortune.cnn.com/2008/03/25/youtube-looks-for-the-money-clip, March 2008.

[3] Alvestrand, H. T. Overview: Real Time Protocols for Brower-based Applications - IETF, June 2012. Work in progress. Expires December, 2012 http://datatracker.ietf.org/doc/draft-ietf-rtcweb-overview/.

[4] Bakker, A., Petrocco, R., Dale, M., J., G., V., G., D., R., and J., P. Online video using bittorrent and html5 applied to wikipedia. In *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on* (aug. 2010), pp. 1 –2.

[5] Bergkvist, A., Burnett, D. C., Jennings, C., and Narayanan, A. WebRTC 1.0: Real-Time communication Between Browsers, February 2012. Work in progress. http://www.w3.org/TR/webrtc/.

[6] Berners-Lee, T. WorldWideWeb, the first Web client. http://www.w3.org/People/Berners-Lee/WorldWideWeb.html.

[7] Berners-Lee, T., and Fischetti, M. *Weaving the Web.* Harper-Collins, 2000, p. 23.

[8] Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A., and Singh, A. Splitstream: High-bandwidth content distribution in cooperative environments. In *Proceedings of IPTPS03* (Feb 2003).

[9] CISCO Visual Networking Index. Entering the Zettabyte Era. Tech. rep., CISCO, June 2011.

[10] Cohen, B. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems* (2003), vol. 6, pp. 68–72.

[11] Cohen, B. The BitTorrent Protocol Specification, Version 11031. `http://www.bittorrent.org/beps/bep_0003.html`, Jan. 2008.

[12] Crockford, D. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.

[13] Fette, I., and Melnikov, A. The WebSocket Protocol. RFC 6455 (Proposed Standard), Dec. 2011.

[14] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[15] Handley, M., and Jacobson, V. SDP: Session Description Protocol. RFC 2327 (Proposed Standard), Apr. 1998. Obsoleted by RFC 4566, updated by RFC 3266.

[16] Hégaret, P. L., Hors, A. L., and Stenback, J. Document Object Model (DOM) Level 2 HTML Specification. W3C recommendation, W3C, Jan. 2003. `http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109`.

[17] Hickson, I. The WebSocket API- W3C Working Draft, May 2012. Work in progress. `http://www.w3.org/TR/websockets/`.

[18] Hickson, I. Web Workers - W3C Candidate Recommendation, May 2012. Work in progress. `http://www.w3.org/TR/workers/`.

[19] Jacobs, I., Raggett, D., and Hors, A. L. HTML 4.01 Specification. W3C recommendation, W3C, Dec. 1999. `http://www.w3.org/TR/1999/REC-html401-19991224`.

[20] Jennings, C., Rosenberg, J., Uberti, J., and Jesup, R. RTCWeb Offer/Answer Protocol (ROAP) - IETF, October 2011. Work in progress. Expires May, 2012 `http://tools.ietf.org/html/draft-jennings-rtcweb-signaling-01`.

[21] Jesup, R., Loreto, S., and Tuexen, M. RTCWeb Datagram Connection, March 2012. Work in progress. Expires September, 2012 `http://tools.ietf.org/html/draft-ietf-rtcweb-data-channel-00`.

[22] Klensin, J. Simple Mail Transfer Protocol. RFC 2821 (Proposed Standard), Apr. 2001. Obsoleted by RFC 5321, updated by RFC 5336.

[23] Klensin, J. Simple Mail Transfer Protocol. RFC 5321 (Draft Standard), Oct. 2008.

[24] Kreitz, G., and Niemela, F. Spotify – large scale, low latency, p2p music-on-demand streaming. In *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on* (aug. 2010), pp. 1 –10.

[25] Loreto, S., Saint-Andre, P., Salsano, S., and Wilkins, G. Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. RFC 6202 (Informational), Apr. 2011.

[26] Mahy, R., Matthews, P., and Rosenberg, J. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766 (Proposed Standard), Apr. 2010.

[27] Mehta, N., Sicking, J., Graff, E., Popescu, A., and Orlow, J. Indexed Database API - W3C Working Draft , May 2012. Work in progress. `http://www.w3.org/TR/IndexedDB/`.

[28] Mol, J. *"Free-riding Resilient Video Streaming in Peer-to-Peer Networks"*. PhD thesis, Department of Software Technology, Delft University of Technology, Delft, Jan 2010.

[29] Mol, J., Bakker, A., Pouwelse, J., Epema, D., and Sips, H. "The design and deployment of a bittorrent live video streaming solution". In *IEEE International Symposium on Multimedia, 2009* (Dec 2009).

[30] Mol, J., Pouwelse, J., Meulpolder, M., Epema, D., and Sips, H. "Give-to-Get: Free-riding Resilient Video-on-demand in P2P Systems". In *Multimedia Computing and Networking conference (Proceedings of SPIE Vol. 6818)* (Jan 2008).

[31] Moore, G. E. Cramming more components onto integrated circuits. *Electronics Magazine, volume 38* (April 1965).

[32] P2P Next News. Eu sponsors p2p tv with 14m euros, 2008. `http://www.p2p-next.org/index.php?page=news&id=FFC9AD9FC7E0072EA5D96ED4E1D1636E` (in English). Accessed 19.2.2008.

[33] Pilgrim, M. *HTML5: Up and Running.* O'Reilly Media, Inc, 2010.

[34] Postel, J. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[35] Pouwelse, J., Garbacki, P., Epema, D., and Sips, H. The bittorrent p2p file-sharing system: Measurements and analysis. In *Peer-to-Peer Systems IV*, M. Castro and R. van Renesse, Eds., vol. 3640 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, pp. 205–216. 10.1007/1155898919.

[36] Pouwelse, J. A., Garbacki, P., Wang, J., and Bakker, A. Tribler: a social-based peer-to-peer system. In *Concurrency and Computation: Practice and Experience* (feb. 2008), vol. 20, pp. 127 –138.

[37] Ranganathan, A., and Sicking, J. File API - W3C Working Draft, October 2011. Work in progress. `http://www.w3.org/TR/FileAPI/`.

[38] Rescorla, E. HTTP Over TLS. RFC 2818 (Informational), May 2000. Updated by RFC 5785.

[39] Rescorla, E. Security Considerations for RTC-Web, October 2011. Work in progress. Expires December, 2012 `http://tools.ietf.org/html/draft-ietf-rtcweb-security-01`.

[40] Rescorla, E., and Modadugu, N. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), Jan. 2012.

[41] Ripeanu, M. Peer-to-peer architecture case study: Gnutella network. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on* (aug 2001), pp. 99 –100.

[42] Rivest, R. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), Apr. 1992. Updated by RFC 6151.

[43] Rosenberg, J. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245 (Proposed Standard), Apr. 2010. Updated by RFC 6336.

[44] Rosenberg, J., Mahy, R., Matthews, P., and Wing, D. Session Traversal Utilities for NAT (STUN). RFC 5389 (Proposed Standard), Oct. 2008.

[45] Rosenberg, J., and Schulzrinne, H. An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264 (Proposed Standard), June 2002. Updated by RFC 6157.

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

[47] S. Androutsellis-Theotokis and D. Spinellis. A survey of content distribution technologies. acm computing surveys, vol. 36, no. 4, December 2004.

[48] Srisuresh, P., and Holdrege, M. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663 (Informational), Aug. 1999.

[49] Stewart, R. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), Sept. 2007. Updated by RFCs 6096, 6335.

[50] Suzuki, Y., and Ogashiwa, N. Real-time web communication by using xmpp jingle, Feb 2012. Work in progress. Expires July, 2012 `http://tools.ietf.org/html/draft-suzuki-rtcweb-jingle-web-00`.

[51] Swanson, B., and Gilder, G. Estimating the Exaflood. Tech. rep., Discovery Institute's Technology and Democracy Project, Januray 2009.

[52] Topf, J. The HTML Form Protocol Attack, 2001. `http://www.remote.org/jochen/sec/hfpa/hfpa.pdf`.

[53] Uberti, J., and Jennings, C. Javascript Session Establishment Protocol - IETF, June 2012. Work in progress. Expires December, 2012 `http://tools.ietf.org/html/draft-ietf-rtcweb-jsep-01`.

[54] van Kesteren, A., and Hickson, I. Offline Web Applications - Offline Application Caching APIs, May 2008. Work in progress. `http://www.w3.org/TR/offline-webapps/#offline`.

[55] Wallsten, K. "Yes We Can": How Online Viewership, Blog Discussion, Campaign Statements, and Mainstream Media Coverage Produced a Viral Video Phenomenon. *Journal of Information Technology & Politics 7*, 2-3 (2010), 163–181.

[56] Wugofski, T., Stark, P., Ishikawa, M., Baker, M., Yamakami, T., and Matsui, S. XHTML^{TM} Basic 1.1. W3C recommendation, W3C, July 2008. `http://www.w3.org/TR/2008/REC-xhtml-basic-20080729`.

[57] Xu, D., Hefeeda, M., Hambrusch, S., and Bhargava, B. On peer-to-peer media streaming. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on* (2002), pp. 363 – 371.

Appendix A

HTML5 JavaScript Source Code

A.1 Video hashing & storage

```
<html>
<head>
    <title>Video on Demand using P2P & HTML5</title>
    <link href="../scripts/videojs/video-js.min.css" rel="stylesheet" type="text/css">
    <script type="text/javascript" src="../scripts/jquery/jquery-1.7.1.js">
    </script>
    <script type="text/javascript" src="../scripts/videojs/video.min.js">
    </script>
    <script type="text/javascript" src="../scripts/md5/webtoolkit.md5.js">
    </script>
    <!--The javascript libraries are loaded above-->
    <script type="text/javascript">
        var SPACE_IN_BYTES = 0;
        var CHUNK_SIZE = 5 * 1024 * 1024;//5MB
        var MAX_FILE_SIZE = 67108864;//64MB
        var SLICE_SIZE = 65536;//64kB
        localStorage["start"] = 50; //how many slices downloaded before video playback
        var GRANTED_BYTES;
        var movie_object = {
            "file_md5_array": [],
            "slices": 0,
            "fileSize": 0,
            "fileName": "",
            "fileURL": "",
            "fileMimeType": "",
            "fileDuration": "",
            "fileID": ""
        };

        var torrent = {}; //Current torrent being downloaded

        function clearMovie_object() {
            movie_object.file_md5_array = [];
            movie_object.slices = 0;
            movie_object.fileName = "";
            movie_object.fileURL = "";
            movie_object.fileMimeType = "";
            movie_object.fileID = "";
            movie_object.fileDuration = "";
        }
        var md5_complete_counter_counter;

        var movie_list = {};
        var indexedDB = window.indexedDB || window.webkitIndexedDB || window.mozIndexedDB;
        if ('webkitIndexedDB' in window) {
            window.IDBTransaction = window.webkitIDBTransaction;
            window.IDBKeyRange = window.webkitIDBKeyRange;
        }
        movie_list.indexedDB = {};
        movie_list.indexedDB.db = null;
        movie_object.fileURL
        movie_list.indexedDB.onerror = function (e) {
            console.log(e);
        };
        function errorHandlerF(e) {
            var msg = '';

            switch (e.code) {
                case FileError.QUOTA_EXCEEDED_ERR:
                    msg = 'QUOTA_EXCEEDED_ERR';
                    break;
                case FileError.NOT_FOUND_ERR:
                    msg = 'NOT_FOUND_ERR';
                    break;
                case FileError.SECURITY_ERR:
                    msg = 'SECURITY_ERR';
                    break;
                case FileError.INVALID_MODIFICATION_ERR:
                    msg = 'INVALID_MODIFICATION_ERR';
                    break;
                case FileError.INVALID_STATE_ERR:
                    msg = 'INVALID_STATE_ERR';
                    break;
                default:
                    msg = 'Unknown Error';
                    break;
            };

            console.log('Error: ' + msg);
        }

        $(document).ready(function () {
            movie_list.indexedDB.open();
            populateTorrents();
            $('#torrents').delegate('input', 'click', function (e) {
                $.get("http://html5p2p-1.cs.hut.fi/tracker/GetTorrentServlet",
                    "torrent_id=" + this.id,
                    function (data) {
                        torrent = JSON.parse(data);
                        console.log("Torrent details", torrent);

                        var data = [];
                        for (var i = 0; i < CHUNK_SIZE; i++) {
                            data.push('0');
                        }//Created a empty slice of '0's
                        var bb = new WebKitBlobBuilder();
                        bb.append(data.join(''));
                        var blob = bb.getBlob(torrent.fileMimeType);

                        var last_slice_size = torrent.fileSize % CHUNK_SIZE;
                        data = [];
                        for (var i = 0; i < last_slice_size; i++) {
                            data.push(’0’);
                        }//Created a empty slice of ’0’s

                        var bb_last = new Blob();
                        bb_last.append(data.join(’’));
                        var last_blob = bb_last.getBlob(torrent.fileMimeType);

                        console.log(’Generated ’ + SLICE_SIZE + ’byte chunk’);
                        window.webkitRequestFileSystem(window.TEMPORARY,
                            torrent.fileSize,
                            function (fs) {
                                fs.root.getFile(torrent.fileName, {
                                    create: true,
                                    exclusive: false
                                },
                                function (fileEntry) {
                                    fileEntry.remove(function () {
                                        console.log(’File removed.’);
                                    }, errorHandlerF);

                                    fs.root.getFile(torrent.fileName, {
                                        create: true,
                                        exclusive: true
                                    },
                                    function (fileEntry) {
                                        var count = 0;
                                        fileEntry.createWriter(function (fw) {
                                            fw.onwriteend = function (e) {
                                                if (fw.length < torrent.fileSize) {
                                                    if (fw.length < torrent.fileSize - last_slice_size) {
                                                        fw.write(blob);
                                                        console.log("SLICE");
                                                    } else {
                                                        fw.write(last_blob);
                                                        console.log("END");
                                                    }
                                                } else {
                                                    console.log(’Ready to download;’);
                                                    $(’#process’).append(’<progress id="download_progress"/>’);
                                                    console.log(’Downloading File...’);
                                                    $(’#download_progress’).attr({
                                                        ’value’: ’0’,
                                                        ’max’: torrent.slices,
                                                    });
                                                    $(’#process’).append(’<progress id="md5_check_progress"/>’);
                                                    console.log(’MD5 verification...’);
                                                    $(’#md5_check_progress’).attr({
                                                        ’value’: ’0’,
                                                        ’max’: torrent.slices,
                                                    });
                                                    $(’#process’).append(’<progress id="slice_write_progress"/>’);
                                                    console.log(’Writing slices...’);
                                                    $(’#slice_write_progress’).attr({
                                                        ’value’: ’0’,
                                                        ’max’: torrent.slices,
                                                    });
                                                    torrent.file_download_status_array = [];
                                                    for (var i = 0; i < torrent.slices; i++) {
                                                        torrent.file_download_status_array[i] = new Boolean(0);
                                                        downloadPiece(i, fileEntry);
                                                    }
                                                }
                                            };
                                            fw.onerror = function (e) {
                                                console.log('Write failed: ' + e.toString());
                                            };
                                            fw.write(blob);
                                        }, errorHandlerF);
                                    }, errorHandlerF);
                                }, errorHandlerF);
                            }, errorHandlerF);
                        });
                    });

            function downloadPiece(i, fileEntry) {
                var start_time = new Date().getTime();
                var xhr0 = new XMLHttpRequest();
                xhr0.open('GET',
                    'http://html5p2p-1.cs.hut.fi/tracker/GetVideoPieceServlet?slice_no=' +
                    i + '&file_name=' + torrent.fileName,
                    function (e) {
                        console.log(e);
                    },
                    null,
                    null);
                xhr0.responseType = 'arraybuffer';
                xhr0.onload = function (e) {
                    var bb = new window.WebKitBlobBuilder();
                    bb.append(xhr0.response);
                    var reader = new FileReader();
                    reader.onloadend = function (e) {
                        var worker = new Worker('../scripts/md5/md5_check.js');
                        var input = {
                            'slice': i,
                            'plain_text': reader.result
                        };

                        worker.postMessage(input);
                        worker.addEventListener('message', function (e) {
                            $('#md5_check_progress').val($('#md5_check_progress').val() + 1);
                            var md5 = e.data.md5;
                            if (md5 === torrent.file_md5_array[i]) {
                                fileEntry.createWriter(function (fileWriter) {
                                    fileWriter.onerror = function (e) {
                                        console.log('Error' + e);
                                    };
                                    fileWriter.onwriteend = function (e) {
                                        $('#slice_write_progress').val($('#slice_write_progress').val() + 1);
                                        torrent.file_download_status_array[i] = Boolean(1);

                                        if (i === 0) {
                                            var speed = SLICE_SIZE / (new Date().getTime() - start_time);
                                            var start = Math.round((((torrent.fileSize / speed) -
                                                (torrent.fileDuration * 1000)) *
                                                speed) / SLICE_SIZE);
                                            if (start > localStorage["start"]) {
                                                localStorage["start"] = start;
                                            }
                                            console.log('Start', localStorage["start"],
                                                'size', torrent.fileSize,
                                                'speed', speed, 'duration',
                                                torrent.fileDuration * 1000,
                                                'Slice', SLICE_SIZE);
                                        } else if (i + 1 === torrent.slices) {
                                            console.log('Completed', torrent.fileID);
                                            openPlayer(fileEntry.toURL(), torrent.fileMimeType);
                                            setTimeout(function () {
                                                $('#slice_write_progress').remove();
                                                $('#md5_check_progress').remove();
                                                $('#download_progress').remove();

                                            }, 2000);
                                        } else if (i == localStorage["start"]) {

                                        }
                                    };

                                    //move the file pointer to the EOF for APPENDING
                                    fileWriter.seek(i * SLICE_SIZE);
                                    fileWriter.write(bb.getBlob(torrent.fileMimeType));
                                }, errorHandlerF);
                            } else {
                                console.log(md5, "re attempt", i);
                            }
                        }, false);
                    };
                    var blob = bb.getBlob(torrent.fileMimeType);
                    reader.readAsText(blob);
                    $('#download_progress').val($('#download_progress').val() + 1);
                };
                xhr0.send();
            }

            $('#File1').bind('change', function (evt) {
                $('#File1').attr("disabled", true);
                $('#process').html('');
                $('#output').html('');
                $('#db').html('');

                movie_object.fileSize = $('#File1')[0].files[0].size;
                movie_object.fileMimeType = $('#File1')[0].files[0].type;
                if (movie_object.fileMimeType.substring(0, 5) == 'video' &&
                    movie_object.fileSize < MAX_FILE_SIZE) { //64MB
                    SPACE_IN_BYTES += movie_object.fileSize;
                    handleFileSelect(evt);
                    movie_object.slices = movie_object.fileSize / SLICE_SIZE;

                    movie_object.slices = Math.ceil(movie_object.slices);
                    $('#process').append('<progress id="md5_progress"/>');
                    console.log('Calculating MD5...');
                    md5_complete_counter = 0;
                    $('#md5_progress').attr({
                        'value': md5_complete_counter,
                        'max': movie_object.slices
                    });
                    console.log(movie_object.slices);

                    for (var i = 0; i < movie_object.slices; i++) {
                        var fnc = function (y) { return function () { readBlob(y); } }(i);
                        setTimeout(fnc, 90 * i);
                    }

                } else {
                    $('#output').text('Since the slice API is not yet standardized, ' +
                        'for now we handle only mp4 videos of 64MB and less..');
                    $('#File1').removeAttr("disabled");
                    setTimeout(function () {
                        $('#output').text('');
                    }, 4000);
                }
            });
            $('#drop_zone').bind('drop', handleFileSelect, false);
            $('#drop_zone').bind('dragover', function (evt) {
                handleDragOver(evt);
            }, false);

        });

        movie_list.indexedDB.open = function () {
            var request = indexedDB.open("movie_records");

            request.onsuccess = function (e) {
                var v = "7.00";
                movie_list.indexedDB.db = e.target.result;
                var db = movie_list.indexedDB.db;
                // We can only create Object stores in a setVersion transaction;
                if (v != db.version) {
                    var setVrequest = db.setVersion(v);

                    // onsuccess is the only place we can create Object Stores
                    setVrequest.onerror = movie_list.indexedDB.onerror;
                    setVrequest.onsuccess = function (e) {
                        if (db.objectStoreNames.contains("movie_record")) {
                            db.deleteObjectStore("movie_record");
                        }
                        var store = db.createObjectStore("movie_record",
                        {
                            keyPath: "fileID"
                        });
                        movie_list.indexedDB.getAllMovieItems();
                    };
                }
                else {
                    movie_list.indexedDB.getAllMovieItems();
                }
            };
            request.onerror = movie_list.indexedDB.onerror;
        }

        movie_list.indexedDB.getAllMovieItems = function () {
            $('#db').html('');

            var db = movie_list.indexedDB.db;
            var trans = db.transaction(["movie_record"], IDBTransaction.READ_WRITE);
            var store = trans.objectStore("movie_record");
            SPACE_IN_BYTES = 0;
            // Get everything in the store;
            var keyRange = IDBKeyRange.lowerBound(0);
            var cursorRequest = store.openCursor(keyRange);
            cursorRequest.onsuccess = function (e) {
                var result = e.target.result;
                if (!!result == false)
                    return;
                renderMovie(result.value);
                result.continue();
            };

            cursorRequest.onerror = movie_list.indexedDB.onerror;
        };

        movie_list.indexedDB.viewMovie = function (id) {

            $('#db').html('');

            var db = movie_list.indexedDB.db;
            console.log('IDBTransaction.READ_WRITE', IDBTransaction.READ_WRITE);
            var trans = db.transaction(["movie_record"], IDBTransaction.READ_WRITE);

            var store = trans.objectStore("movie_record");

            // Get everything in the store;
            var keyRange = IDBKeyRange.lowerBound(0);
            var cursorRequest = store.openCursor(keyRange);
            cursorRequest.onsuccess = function (e) {
                var result = e.target.result;

                if (!!result == false)
                    return;
                if (result.value.fileID === id) {
                    console.log(result.value.fileURL);
                    openPlayer(result.value.fileURL, result.value.fileMimeType);
                }
                result.continue();
            };

            movie_list.indexedDB.getAllMovieItems();
            cursorRequest.onerror = movie_list.indexedDB.onerror;
        };

        movie_list.indexedDB.deleteMovie = function (id) {
            var db = movie_list.indexedDB.db;
            var trans = db.transaction(["movie_record"], IDBTransaction.READ_WRITE);
            var store = trans.objectStore("movie_record");

            var request = store.delete (id);

            request.onsuccess = function (e) {
                movie_list.indexedDB.getAllMovieItems();
            };

            request.onerror = function (e) {
                console.log("Error Adding: ", e);
            };
        };

        function renderMovie(row) {
            var li = document.createElement("li");
            var v = document.createElement("a");
            var d = document.createElement("a");
            var p = document.createElement("a");
            var u = document.createElement("a");
            var t = document.createTextNode(row.text);
            SPACE_IN_BYTES += row.fileSize;
            v.addEventListener("click", function () {
                movie_list.indexedDB.viewMovie(row.fileID);
            }, false);
            d.addEventListener("click", function () {
                movie_list.indexedDB.deleteMovie(row.fileID);
                window.requestFileSystem = window.requestFileSystem || window.webkitRequestFileSystem;
                window.requestFileSystem(window.PERSISTENT, GRANTED_BYTES, function (fs) {
                    fs.root.getFile(row.fileName, { create: false }, function (fileEntry) {
                        fileEntry.remove(function () {
                            console.log(row.fileName, ' File removed.');
                        }, errorHandlerF);
                    }, errorHandlerF);
                }, errorHandlerF);
            }, false);
            p.addEventListener("click", function () {
                console.log("Uploading torrent to server");
                $.post("http://html5p2p-1.cs.hut.fi/tracker/PublishServlet", "torrent=" +
                    JSON.stringify(row), function (data) {
                    console.log("Server returned", JSON.parse(data));
                    populateTorrents();
                    //li.appendChild(u);
                    //£('a', '#db').css('cursor', 'pointer');
                });
            }, false);

            v.textContent = " [View]";
            d.textContent = " [Del]";
            p.textContent = " [Publish]";
            //          u.textContent = " [Upload]";
            li.appendChild(v);
            li.appendChild(p);
            li.appendChild(d);
            li.appendChild(t);


            $('#db').append(li)
            $('a', '#db').css('cursor', 'pointer');
        }

        function populateTorrents() {
            $('#torrents').html("");
            $.get("http://html5p2p-1.cs.hut.fi/tracker/GetTorrentsServlet", "", function (data) {
                $('#torrents').html(data);
            });
        }

        function renameFileAndAddToDb() {
            var fileName = movie_object.fileName;
            var fileMimeType = movie_object.fileMimeType;
            var fileDuration = movie_object.fileDuration;
            movie_object.fileName = "";
            movie_object.fileURL = "";


            var extension = movie_object.fileMimeType.substring(movie_object.fileMimeType.lastIndexOf('/') + 1,
                movie_object.fileMimeType.length);
            movie_object.fileMimeType = "";
            movie_object.fileDuration = "";
            var unique_movie_object_string = JSON.stringify(movie_object);
            movie_object.fileID = MD5(unique_movie_object_string, unique_movie_object_string.length);
            movie_object.fileName = movie_object.fileID + "." + extension;
            movie_object.fileMimeType = fileMimeType;
            movie_object.fileDuration = fileDuration;
            window.requestFileSystem = window.requestFileSystem || window.webkitRequestFileSystem;
            window.requestFileSystem(window.PERSISTENT, GRANTED_BYTES, function (fs) {
                rename(fs.root, fileName, movie_object.fileName);
                function rename(cwd, src, newName) {
                    cwd.getFile(src, {}, function (fileEntry) {
                        fileEntry.moveTo(cwd, newName);
                        var fileURL = fileEntry.toURL();
                        movie_object.fileURL = fileURL.substring(0,
                            fileURL.lastIndexOf('/')) +
                            '/' + newName;
                        var data = $.extend(true, {}, movie_object);
                        data.text = fileName;
                        console.log("Renamed ", fileName, " to ", newName);
                        movie_list.indexedDB.addMovie(data);
                    }, errorHandlerF);
                }
            }, errorHandlerF);
        }

        movie_list.indexedDB.addMovie = function (data) {
            var db = movie_list.indexedDB.db;
            var trans = db.transaction(["movie_record"], IDBTransaction.READ_WRITE);
            var store = trans.objectStore("movie_record");

            var request = store.put(data);
            request.onsuccess = function (e) {
                clearMovie_object();
                $('#File1').removeAttr("disabled");
                movie_list.indexedDB.getAllMovieItems();
            };

            request.onerror = function (e) {
                console.log("Error Adding: ", e);
            };
        };

        function openPlayer(url, mimeType) {
            $('#output').html('');
            var loaded_video = Date.now().toString();
            $('#output').append('<video id="' + loaded_video + '"><source src="' +
                url + '" type="' + mimeType + '"></video>');
            $('video', '#output').addClass('video-js vjs-default-skin');
            $('video', '#output').attr({
                'width': '640',
                'height': '264',
                'controls': 'true',
                'preload': 'auto',
                'autoplay': 'true',
            });

            _V_(loaded_video).ready(function () {
                var myPlayer = this;
                myPlayer.addEvent("loadstart", function (e) {
                    console.log('Player', 'loadstart');
                });
                myPlayer.addEvent("play", function (e) {
                    console.log('Player', 'play');
                });
                myPlayer.addEvent("error", function (e) {
                    console.log('Player', 'Error @ ', myPlayer.currentTime());
                    openPlayer(url, mimeType);
                });
                myPlayer.addEvent("progress", function (e) {
                    console.log('Player', 'progress', e);
                });

                myPlayer.addEvent("durationchange", function (e) {
                    movie_object.fileDuration = myPlayer.duration();
                    console.log('Player', 'durationchange', e, 'Duration', myPlayer.duration());
                });
            });
        }


        function handleFileSelect(evt) {
            evt.stopPropagation();
            evt.preventDefault();

            movie_object.fileName = $('#File1')[0].files[0].name;
            window.requestFileSystem = window.requestFileSystem || window.webkitRequestFileSystem;

            window.webkitStorageInfo.requestQuota(window.PERSISTENT,
                SPACE_IN_BYTES + MAX_FILE_SIZE,
                function (grantedBytes) {
                    GRANTED_BYTES = grantedBytes;
                    window.requestFileSystem(window.PERSISTENT, GRANTED_BYTES,
                        function onInitFs(fs) {
                            fs.root.getFile(movie_object.fileName, { create: true },
                                function (fileEntry) {
                                    // Create a FileWriter object for our FileEntry (log.txt).
                                    fileEntry.createWriter(function (fileWriter) {
                                        fileWriter.onerror = function (e) {
                                            console.log('Write failed: ' + e);
                                        };
                                        fileWriter.onwritestart = function (e) {
                                            // Reset progress indicator on new file selection.
                                            $('#process')
                                                .append('<progress id="file_select_progress"' +
                                                ' value="0" max="0"/>');
                                            console.log('Copying File...');
                                            $('#file_select_progress').attr({
                                                'value': '0',
                                                'max': e.total,
                                            });
                                        }
                                        fileWriter.onprogress = function (e) {
                                            $('#file_select_progress').attr({
                                                'value': e.progress
                                            });
                                        };
                                        fileWriter.onwriteend = function (e) {
                                            // Ensure that the progress bar displays 100% at the end.
                                            $('#file_select_progress').attr({
                                                'value': '1',
                                                'max': '1',
                                            });
                                            movie_object.fileURL = fileEntry.toURL();
                                            openPlayer(movie_object.fileURL, movie_object.fileMimeType);
                                            setTimeout(function () {
                                                $('#file_select_progress').remove();
                                            }, 2000);
                                        };
                                        // Create a new Blob
                                        var bb = new window.WebKitBlobBuilder();// in Chrome 12.
                                        if (evt.type === 'change') {
                                            bb.append(evt.target.files[0]);
                                        } else {
                                            bb.append(evt.dataTransfer.files[0]);
                                        }
                                        fileWriter.write(bb.getBlob(movie_object.fileMimeType));
                                    }, errorHandlerF);
                                }, errorHandlerF);
                        }, errorHandlerF);
                }, function (e) {
                    console.log('requestQuotaError', e);
                });
        }

        function readBlob(piece_pos) {
            var file = $('#File1')[0].files[0];

            var opt_startByte = piece_pos * SLICE_SIZE;
            if (opt_startByte > file.size - 1) {
                opt_startByte = -1;
            }
            var opt_stopByte = opt_startByte + SLICE_SIZE - 1;
            if (opt_stopByte > file.size - 1) {
                opt_stopByte = file.size - 1;
            }

            var start = opt_startByte;
            var stop = opt_stopByte;

            var reader = new FileReader();
            // If we use onloadend, we need to check the readyState.
            reader.onloadend = function (evt) {

                if (evt.target.readyState == FileReader.DONE) { // DONE == 2
                    var worker = new Worker('../scripts/md5/md5_check.js');
                    var input = {
                        'slice': piece_pos,
                        'plain_text': evt.target.result
                    };
                    //console.log(input.plain_text);
                    worker.postMessage(input);
                    worker.addEventListener('message', function (e) {
                        movie_object.file_md5_array[e.data.slice] = e.data.md5;
                        console.log(e.data.md5);
                        $('#md5_progress').attr({
                            'value': ++md5_complete_counter,
                        });
                        if (md5_complete_counter === movie_object.slices) {
                            renameFileAndAddToDb();
                            setInterval(function () {
                                $('#md5_progress').remove();
                            }, 2000);
                        }
                    }, false);
                }
            };

            if (file.slice) {
                var blob = file.slice(start, stop + 1);
            } else if (file.mozSlice) {
                var blob = file.mozSlice(start, stop + 1);
            } else if (file.webkitSlice) {
                var blob = file.webkitSlice(start, stop + 1);
            } else {
                console.log('Problem');
            }

            reader.readAsText(blob, 'UTF-8');
        }

        function handleDragOver(evt) {
            evt.stopPropagation();
            evt.preventDefault();
            evt.dataTransfer.dropEffect = 'copy'; // Explicitly show this is a copy.
        }

    </script>
</head>
<body>
    <div id="drop_zone">
        <div id="input">
            Select an MP4 Video file, it would get copied into the PERMANENT
            HTML5 FileSystem, and played within the browser:<br>
            <input accept="video/*" id="File1" type="file" />
            <p>
        </div>
        <div id="process"></div>
        <div id="output"></div>
        <div id="db"></div>
        <p>
            <div id="torrents"></div>
    </div>
</body>
</html>
```

A.2 Video conferencing

A.2.1 Tracker implementation - Server-side with node.js

```
#!/usr/bin/env node
var WebSocketServer =  require('websocket').server;
var http = require('http');
var rooms=[];
var connections=[];
var connection_id_counter=0;
var room_id_counter=0;

var server = http.createServer(function(request, response) {
    console.log((new Date()) + ' Received request for ' + request.url);
    response.writeHead(404);
    response.end();
});
server.listen(8080, function() {
    console.log((new Date()) + ' Server is listening on port 8080');
});

wsServer = new WebSocketServer({
    httpServer: server,
    // You should not use autoAcceptConnections for production
    // applications, as it defeats all standard cross-origin protection
    // facilities built into the protocol and the browser.  You should
    // *always* verify the connection's origin and decide whether or not
    // to accept it.
    autoAcceptConnections: false
```

```
});

function originIsAllowed(origin) {
  // put logic here to detect whether the specified origin is allowed.
  console.log('The origin is',origin);
  return true;
}

wsServer.on('request', function(request) {
    if (!originIsAllowed(request.origin)) {
      // Make sure we only accept requests from an allowed origin
      request.reject();
      console.log((new Date()) + ' Connection from origin ' + request.origin + ' rejected.');
      return;
    }

    var con = request.accept('meyn', request.origin);
  var connection = {};
  connection.id = ++connection_id_counter;
  connection.con = con;
  connections.push(connection);

  var msgCONNECTION_ID = {};
  msgCONNECTION_ID.msg_type = 'CONNECTION_ID';
  msgCONNECTION_ID.connection_id = connection.id;
    connection.con.send(JSON.stringify(msgCONNECTION_ID));
    console.log((new Date()) + ' Connection accepted.');

    con.on('message', function(message) {
        if (message.type === 'utf8') {
      processMessageFromClient(con,message.utf8Data);
        }
        else if (message.type === 'binary') {
            console.log('Received Binary Message of ' + message.binaryData.length + ' bytes');
        }
    });
  Array.prototype.contains = function(obj) {
    var i = this.length;
    while (i--) {
        if (this[i] === obj) {
            return true;
        }
    }
    return false;
}
  function processMessageFromClient(con,message){
    var msg = JSON.parse(message);
    console.log(msg.msg_type);
    switch(msg.msg_type)
    {
      case "CREATE_ROOM_REQUEST":
        var msgCREATE_ROOM_RESPONSE ={};
        msgCREATE_ROOM_RESPONSE.msg_type = 'CREATE_ROOM_RESPONSE';
        msgCREATE_ROOM_RESPONSE.room_id = ++room_id_counter;
        msgCREATE_ROOM_RESPONSE.room_key = randomPassword(8);
        var room = {};
        room.room_id = msgCREATE_ROOM_RESPONSE.room_id;
        room.room_key = msgCREATE_ROOM_RESPONSE.room_key;
        room.connection_ids = [];
        rooms.push(room);
        con.send(JSON.stringify(msgCREATE_ROOM_RESPONSE));
      break;
      case "JOIN_ROOM_REQUEST":
        var msgJOIN_ROOM_RESPONSE ={};
        msgJOIN_ROOM_RESPONSE.msg_type = 'JOIN_ROOM_RESPONSE';
        msgJOIN_ROOM_RESPONSE.status = 'Failure';
        for(var i in rooms){
          if(rooms[i].room_id==msg.room_id && rooms[i].room_key==msg.room_key){
            msgJOIN_ROOM_RESPONSE.status = 'Success';
            var cid = connectionId(con);
            if(!rooms[i].connection_ids.contains(cid)){
              rooms[i].connection_ids.push(cid);
            }
            msgJOIN_ROOM_RESPONSE.connection_ids = rooms[i].connection_ids;
            break;
          }
        }
        console.log(rooms);
        con.send(JSON.stringify(msgJOIN_ROOM_RESPONSE));
      break;
      case "PEER_INFO_CLIENT":
      //setTimeout(function(){
        var msgPEER_INFO_SERVER = {};
        msgPEER_INFO_SERVER.msg_type = 'PEER_INFO_SERVER';
        msgPEER_INFO_SERVER.from_connection_id = connectionId(con);
        console.log(msgPEER_INFO_SERVER.from_connection_id,"->",msg.to_connection_id);
        msgPEER_INFO_SERVER.data = msg.message_data;
        for(var i in connections){
          if(connections[i].id==msg.to_connection_id){
            connections[i].con.send(JSON.stringify(msgPEER_INFO_SERVER));
            break;
          }
        }
        //},6000);
      break;
      default:
        console.log('DEFAULT');
    }
  }
  function connectionId(con){
    for(var i in connections){
      var connection = connections[i];
      if(con==connection.con){
        return connection.id;
      }
    }
    return 0;
  }
  function randomPassword(length){
    chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";
    pass = "";
    for(x=0;x<length;x++)
    {
      i = Math.floor(Math.random() * 62);
      pass += chars.charAt(i);
    }
    return pass;
  }
    con.on('close', function(reasonCode, description) {
        console.log((new Date()) + ' Peer ' + con.remoteAddress + ' disconnected.');
    var connection_id = connectionId(con);
    for(var i in connections) {
      var value = connections[i];
      if(connections[i].id===connection_id){
          connections.splice(i,1);
          for(var j in rooms){
            for(var k in rooms[j].connection_ids){
              if(rooms[j].connection_ids[k]==connection_id){
                rooms[j].connection_ids.splice(k,1);
                if(rooms[j].connection_ids.length==0){
                  rooms.splice(j,1);
                }
              }
            }
          }
          break;
        }
      }
    });
});
```

A.2.2 Client side implementation

```
<html>
<head>
    <link rel="canonical" href="client.html" />
    <link rel="StyleSheet" href="css/style.css" type="text/css" media="screen">
    <script type="text/javascript" src="js/jquery-1.7.1.js">
</script>
    <script type="text/javascript">
        var localStream;
        var socket;
        var my_connection_id;
        var channelReady = false;
        var peer_connections = [];
        var remoteVideoHtml = '<video width="50%" height="50%" id="remoteVideoId"' +
              ' autoplay="autoplay" style="opacity: 1; -webkit-transition: opacity 2s;">';
        $(document).ready(function () {
            console.log('Ready');
            getUserMedia();
            $('#create_room').bind('click', function () {
                if (channelReady) {
                    console.log('Sending a CREATE_ROOM_REQUEST via the WebSocket to the server');
                    var msgCREATE_ROOM_REQUEST = {};
                    msgCREATE_ROOM_REQUEST.msg_type = 'CREATE_ROOM_REQUEST';
                    socket.send(JSON.stringify(msgCREATE_ROOM_REQUEST));

                } else {
                    console.log('Web Socket Not Yet established..So cannot create room');
                }
            });
            $('#connect').bind('click', function () {
                if ($('#room_id').val() && $('#room_key').val()) {
                    $('#remote').html('');
                    console.log('Sending a JOIN_ROOM_REQUEST via the WebSocket to the server');
                    var msgJOIN_ROOM_REQUEST = {};
                    msgJOIN_ROOM_REQUEST.msg_type = 'JOIN_ROOM_REQUEST';
                    msgJOIN_ROOM_REQUEST.room_id = $('#room_id').val();
                    msgJOIN_ROOM_REQUEST.room_key = $('#room_key').val();
                    socket.send(JSON.stringify(msgJOIN_ROOM_REQUEST));
                } else {
                    console.log('No room_key and id combination to send to the WebSocket');
                }
            });
        });

        openChannel = function () {
            console.log("Opening channel.");
            socket = new WebSocket('ws://82.130.14.187:8080', 'meyn');
            socket.onopen = function () {
                console.log('Channel opened.');

            };

            socket.onerror = function (error) {
                console.log('Channel error.', error);
            };

            socket.onclose = function () {
                console.log('Channel close.');
            };

            // Log messages from the server
            socket.onmessage = function (e) {
                var msg = JSON.parse(e.data);
                switch (msg.msg_type) {
                    case "CONNECTION_ID":
                        my_connection_id = msg.connection_id;
                        channelReady = true;
                        break;
                    case "CREATE_ROOM_RESPONSE":
                        console.log('Recieved a CREATE_ROOM_RESPONSE via the WebSocket');
                        $('#room_id').val(msg.room_id);
                        $('#room_key').val(msg.room_key);
                        break;
                    case "JOIN_ROOM_RESPONSE":
                        console.log('Recieved a JOIN_ROOM_RESPONSE via the WebSocket');
                        if (msg.status == 'Success') {
                            for (var i in msg.connection_ids) {
                                if (msg.connection_ids[i] != my_connection_id) {
                                    var peer_connection = {};
                                    peer_connection.connection_id = msg.connection_ids[i];
                                    peer_connection.pc =
                                        createPeerConnection(peer_connection.connection_id, true);
                                    peer_connections.push(peer_connection);
                                    $('#remote').prepend(remoteVideoHtml
                                        .replace('remoteVideoId', 'peer' +
                                        peer_connection.connection_id));
                                }
                            }
                        } else {
                            console.log('JOIN_ROOM_RESPONSE : Status:', msg.status);
                        }
                        break;
                    case "PEER_INFO_SERVER":
                        var new_connection = true;
                        console.log('Recieved a PEER_INFO_SERVER via the WebSocket from ',
                            msg.from_connection_id, msg);
                        if (msg.from_connection_id != my_connection_id) {
                            for (var i in peer_connections) {
                                if (peer_connections[i].connection_id == msg.from_connection_id) {
                                    new_connection = false;
                                    break;
                                }
                            }
                            if (new_connection) {
                                console.log('New Peer Connection');
                                var peer_connection = {};
                                peer_connection.connection_id = msg.from_connection_id;
                                peer_connection.pc = createPeerConnection(peer_connection.connection_id,
                                    false);
                                peer_connections.push(peer_connection);
                                $('#remote').prepend(remoteVideoHtml.replace('remoteVideoId', 'peer' +
                                    peer_connection.connection_id));
                            }
                            //Now process the SDP JSON Blob received
                            for (var i in peer_connections) {
                                if (peer_connections[i].connection_id == msg.from_connection_id) {
                                    peer_connections[i].pc.processSignalingMessage(msg.data);
                                    break;
                                }
                            }
                        } else {
                            console('Why is the server sending me my own message???');
                        }

                        break;
                    default:
                        console.log('DEFAULT');
                }
            };
        }
        createPeerConnection = function (connection_id, initiator) {
            if (initiator) { console.log('INITIATOR'); }
            //var STUN_OR_TURN = "TURN 193.234.219.124:3478";
            var STUN_OR_TURN = "STUN stun.l.google.com:19302";
            if (typeof (webkitPeerConnection00) === 'undefined') {
                alert("webkitPeerConnection00");
            }
            try {
                pc = new webkitDeprecatedPeerConnection(STUN_OR_TURN, function (message) {
                    onSignalingMessage(connection_id, message);
                });
                console.log("Created webkitDeprecatedPeerConnnection with config ", STUN_OR_TURN);
            } catch (e) {
                try {
                    console.log("Failed to create webkitDeprecatedPeerConnection, exception: " + e.message);
                    pc = new webkitPeerConnection00(STUN_OR_TURN, function (message) {
                        onSignalingMessage(connection_id, message);
                    });
                    console.log("Created webkitPeerConnection00 with config ", STUN_OR_TURN);

                } catch (e) {
                    console.log("Failed to create webkitPeerConnection00, exception: " + e.message);
                    try {
                        pc = new webkitPeerConnection(STUN_OR_TURN, function (message) {
                            onSignalingMessage(connection_id, message);
                        });
                        console.log("Created webkitPeerConnnection with config ", STUN_OR_TURN);
                    } catch (e) {
                        console.log("Failed to create webkitPeerConnection, exception: " + e.message);
                        alert("Cannot create PeerConnection object; tried webkitPeerConnection"+
                            " and webkitDeprecatedPeerConnection");
                        return;
                    }
                }
            }
            pc.addStream(localStream);
            pc.onconnecting = function (msg) {
                console.log('onSessionConnecting');
            }
            pc.onopen = function (msg) {
                console.log('onSessionOpened');
            }
            pc.onaddstream = function (event) {
                console.log('onRemoteStreamAdded add the remote peers video stream.');
                var url = webkitURL.createObjectURL(event.stream);
                $('#peer' + connection_id).attr({
                    src: url
                });
            }
            pc.onremovestream = function (msg) {
                console.log('onRemoteStreamRemoved');
            }
            return pc;
        }
        onSignalingMessage = function (connection_id, message) {
            var msgPEER_INFO_CLIENT = {};
            msgPEER_INFO_CLIENT.msg_type = 'PEER_INFO_CLIENT';
            msgPEER_INFO_CLIENT.to_connection_id = connection_id;
            msgPEER_INFO_CLIENT.message_data = message;
            console.log(new Date().getTime(), ": C(", my_connection_id, ")->S(",
                msgPEER_INFO_CLIENT.to_connection_id, ")", msgPEER_INFO_CLIENT);
            socket.send(JSON.stringify(msgPEER_INFO_CLIENT));
        }
        getUserMedia = function () {
            try {
                navigator.webkitGetUserMedia({ video: true, audio: true }, onUserMediaSuccess,
                                       onUserMediaError);
                console.log("Requested access to local media.");
            } catch (e) {
                alert("webkitGetUserMedia() failed. Does your broser support WebRTC?");
                console.log("webkitGetUserMedia failed with exception: " + e.message);
            }
        }
        onUserMediaSuccess = function (stream) {
            console.log("User has granted access to local media.");
            openChannel();
            var url = webkitURL.createObjectURL(stream);
            $('#localVideo').attr({
                src: url
            });
            localStream = stream;
        }
        onUserMediaError = function (error) {
            console.log("Failed to get access to local media. Error code was " + error.code);
            alert("Failed to get access to local media. Error code was " + error.code + ".");
        }

</script>
</head>
<body>

    <div id="container">
        <div id="room_details">
            <input type="button" id="create_room" value="Create room" />
            <label for="room_id_l">Room ID</label>
            <input type="text" id="room_id" />
            <label for="room_key_l">Room Key</label>
            <input type="text" id="room_key" />
            <input type="button" id="connect" value="Connect to room" />
        </div>
        <div id="local">
            <video width="50%" height="50%" id="localVideo"
                autoplay="autoplay" style="opacity: 1;
                -webkit-transition: opacity 2s;">
            </video>
        </div>
        <div id="remote">
        </div>
        <div id="footer">
        </div>
    </div>
    <img id="logo" alt="WebRTC" src="images/webrtc_black_20p.png">
</body>
</html>
```