Unified Communication and WebRTC

Simen Owesen-Lein

Master of Science in Communication Technology
Submission date: Januar 2015
Supervisor: Mazen Malek Shiaa, ITEM

Norwegian University of Science and Technology
Department of Telematics

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Project Description

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Telematics

Unified Communication and WebRTC

by Simen Owesen-Lein

Web Real-Time Communication (WebRTC) offers application developers the ability to write rich, real-time multimedia applications (e.g. video chat) on the web, without requiring plugins, downloads or installs. As such users can simply use web-browsers to make voice/video calls and perform collaborative tasks by sharing desktops, files and whiteboards. On the other side the existing eco-systems for telephony and messaging services allow users perform traditional communication tasks, such as voice (to some extent video) calls and text messaging, over existing infrastructures/devices, such as mobile, fixed and VoIP networks and handsets.

The scope of this task is to investigate how WebRTC capabilities can enhance the existing eco-systems for telephony and messaging services by providing the end-users with advanced communication capabilities. The task will focus on certain use cases for value-added-services. The task will also include a study in different architectures.

To experiment this concept the student will improve a WebRTC client prototype for Chrome and Firefox by which both rich communication scenarios as well integration with an existing telephony and messaging ecosystem will be demonstrated. A whiteboard feature and screensharing will also be used to improve the prototype.

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Abstract

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Telematics

Unified Communication and WebRTC

by Simen Owesen-Lein

Web Real-Time Communication (WebRTC) is a technology enabling web browsers to connect directly to each other, and send information e.g. video or text messages peer-to-peer. This technology is currently being implemented in the major browsers, and is under rapid development.

Unified communications seeks to bridge the gap between different communication protocols, and allows users to communicate across technologies. An existing web application prototype allows users to communicate with audio and video. Users can connect with a web browser using WebRTC or a phone using the telephony network. The goal of this thesis is to improve the functionality of this existing prototype, and experiment with and research WebRTC technology and applications.

The original prototype uses a centralized server based structure. The improvements in this thesis utilizes the peer-to-peer benefits of WebRTC. The improvements include file sharing, whiteboard and screen sharing.

NORGES TEKNISK-NATURVITENSKAPLIGE UNIVERSITET

Sammendrag

Fakultet for informasjonsteknologi, matematikk og elektroteknikk
Institutt for telematikk

Unified Communication og WebRTC

av Simen Owesen-Lein

Web Real-Time Communication (WebRTC) er en teknologi som tilater nettlesere å koble seg direkte opp mot hverandre, og sende informasjon som video eller tekstmeldinger peer-to-peer. Denne teknologien er under implementasjon i de store nettleserene, og endres kontinuerlig.

Unified communications handler om å kombinere forskjellige kommunikasjonsprotokoller, og tilater brukere å kommunisere over forskjellige teknologier. I en eksisterende web-applikasjon har brukere mulighet til å kommunisere med lyd og video. Det er mulig å bruke en nettleser ved hjelp av WebRTC, eller en telefon ved hjelp av telefonnettet. Målet med denne oppgaven er å videreutvikle funksjonaliteten i denne prototypen, og eksperimentere med og undersøke WebRTC teknologi og bruksområder.

Prototypen bruker en sentralisert serverstruktur. Forbedringene lagt frem i denne oppgaven bruker peer-to-peer mulighetene som WebRTC tilbyr. Forbedringene inkluderer fildeling, whiteboard og skjermdeling.

Acknowledgements

Written by Simen Owesen-Lein, at the Department of Telematics, NTNU. Fall 2014

I would like to thank the responsible professor Mazen Shiaa from ITEM and Gintel AS. Also thanks to Xiao Chen from Gintel AS for assisting me with technical issues.

Contents

List of Figures

List of listings

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
Blob	**B**inary **L**arge **O**bject
CPU	**C**entral **P**rocessing **U**nit
DOM	**D**ocument **O**bject **M**odel
HD	**H**igh **D**efinition
HTML	**H**yper**T**ext **M**arkup **L**anguage
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
HTTPS	**H**yper**T**ext **T**ransfer **P**rotocol **S**ecure
ICE	**I**nteractive **C**connectivity **E**stablishment
IETF	**I**nternet **E**ngineering **T**ask **F**orce
IP	**I**nternet **P**rotocol
RTC	**R**eal-**T**ime **C**ommunication
SDP	**S**ession **D**escription **P**rotocol
SIP	**S**ession **I**nitiation **P**rotocol
SMS	**S**hort **M**essage **S**ervice
SRTCP	**S**ecure **R**eal-time **T**ransport **C**ontrol **P**rotocol
SRTP	**S**ecure **R**eal-time **T**ransport **P**rotocol
STUN	**S**ession **T**raversal **U**tilities for **N**AT
TLS	**T**ransport **L**ayer **S**ecurity
TURN	**T**raversal **U**sing **R**elays around **N**AT
W3C	**W**orld **W**ide **W**eb **C**onsortium
WebRTC	**W**eb **R**eal-**T**ime **C**ommunication

1. Introduction

Unified communications seeks to combine different communication platforms into one. This includes but is not limited to instance messages, Short Message Service (SMS), email, conference calls, file sharing, presence updates etc. Unified communications can include all possible communications over a network. The goal is to make the difference in protocols invisible to the user, and allow them to communicate with different technologies by using what is most natural in the setting. A potential use of this is to connect to a service with multiple devices. For example a laptop, and a phone. By using certain rules and logic, the service can determine which device should be active. For example during work hours any calls or communications should be routed to the work computer. When driving home, the user can be reached using their phone. When they come home, email might be more appropriate. This scenario would bridge together email, SMS, regular phone calls and a video conferencing application. The benefit of unified communications is usability. Easy access to communications using multiple platforms could increase productivity in a work setting, or just decreasing the hassle of switching between protocols in a social setting.

As access to the Internet and available bandwidth increases, the demand for communication applications using the Internet increases as well. Several solutions exist today e.g. Skype[1] that enable users to communicate using video and voice. Skype is a stand-alone application. Video conferencing in a web browser is also possible. There are many existing ways to achieve video conferencing using a web browser. What all the existing conferencing tools have in common is the use of third party plugins. Adobe Flash[2] is a commonly used plugin to enable dynamic and expressive web content including video conferencing. The use of Flash is currently declining[3]. With the emergence of HTML5 and other advances in web browsers, third party plugins like Adobe Flash is slowly being replaced by native functionality. WebRTC continues on this path by enabling the use of video conferencing and other peer to peer real-time communication in the browser without plugins. This makes it easier for the end user to use an application if it means they don't have to download additional software to make it work.

On the other hand there is the telephony network where most people already have access. By using just WebRTC and the telephony network, and integrating these into one service, it is possible to create a unified communications platform which enables access for a large portion of the population. One usually has either access to a phone network or access to the Internet. If you are somewhere remote without cell phone coverage, you might still be able to use Internet via a satellite or vice versa. By using only two different platforms the service is easily available and functional for most users out of the box. A prototype was developed in a previous master thesis[4]. This prototype system can host communications between WebRTC and the telephony network.

The goal of this thesis is to continue the development of the existing prototype by adding or improving existing features, and to study and evaluate WebRTC technology in the context of unified communications.

1.1 Method

This section will describe the different phases of the work on this thesis.

1.1.1 Research

It was important to gain a thorough understanding of the different technologies involved in the prototype and WebRTC in general. This was accomplished by reading the master thesis responsible for the prototype[4], and also reading documentation on WebRTC in the form of articles, draft specifications and open source code. This work is documented in chapter 2, Exploring WebRTC and chapter 3, The prototype system.

1.1.2 Use cases

To decide on what improvements to add to the prototype, an approach using use cases was utilized. The use cases provide a story about a possible use of the prototype. This is used to gain a non-technical insight into common uses. By ignoring the technical aspects, it is easier to focus on what is important to the user, and not what is easy or possible to create for the developer. Chapter 4 describes the different use cases that was investigated.

1.1.3 Development

The next step was to decide the different features to implement. The use cases described in chapter 4 provides several potential uses for the application, and states necessary technologies for each case. Three technologies was extracted from these use cases. Namely, screen sharing, whiteboard and file sharing. They were chosen based on frequency in the use cases, usefulness and how interesting they were from an academic aspect.

The development phase is documented in chapter 5, Improving the prototype. This section describes the method behind the design, implementation and testing of the chosen prototype improvements.

The waterfall model[5] is a sequential design process used in planning a software development project. It involves several separate phases that have their own responsibility. For example a design phase and a test phase. This is beneficial if the requirements for the software is clearly defined and unlikely to change.

Incremental development methods[5] are more fluid in the sense that they don't define specific phases responsible for testing and implementation, but focus more on specific features and sub parts of the project. These parts are then completed, and include their own design, implementation and testing. The different phases are usually more intertwined than the waterfall model, and testing occurs while developing, not after. This approach has the benefit of quickly producing visible results. It is also more robust to changes than the waterfall model. If the requirements change, maybe as a result of seeing a demo, it is easier to change individual features and parts than it is to redesign the entire workflow of the project.

These development models are abstract and high level, and there exists a lot of more specific development models. Since the development of the individual features in this thesis was a one man project with limited scope, it was not necessary to adapt a specific full on development scheme to create the improvements. Several of the benefits of choosing an appropriate development model include improved cooperation in the working team and with the client, and this did not apply to my project. Therefore the abstract approach of incremental development was more than sufficient. The individual improvements represent their own sub part of the project, and they all had their own creation phase with design, implementation and testing.

1.2 Thesis structure

Chapter 2 will delve into WebRTC technology on what features are available, and how to use these.

Chapter 3 describes the different parts of the existing prototype system.

Chapter 4 describes three different use cases that are used to justify the prototype improvements.

Chapter 5 details the improvements and features developed for the prototype.

Chapter 6 includes issues, discussion, future work and conclusion.

Appendix A and B include the related source code.

2. Exploring WebRTC

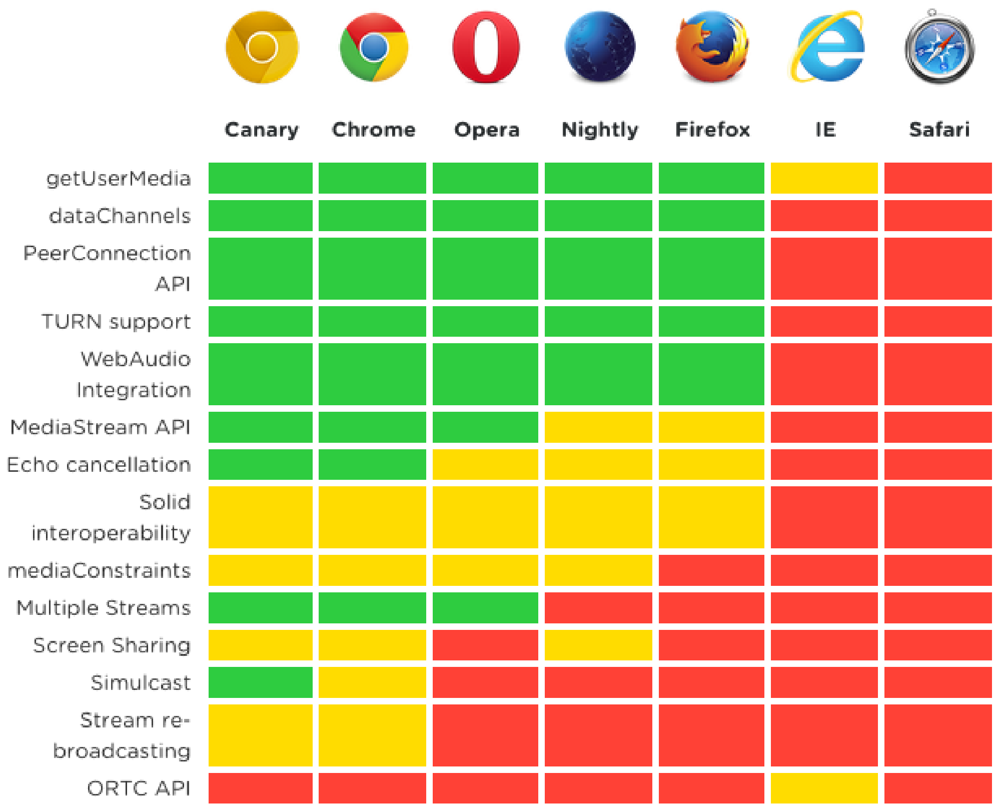
This chapter documents the research and investigation of WebRTC technology. A general description of WebRTC is found in section 2.1. Section 2.2 shows the WebRTC implementation status and support in the major web browsers. The remaining sections of this chapter will go through the sub features of WebRTC that are essential to this thesis and how to use them.

2.1 Description

Web Real-Time Communication (WebRTC) is a proposed API for ECMAScript drafted by the World Wide Web Consortium (W3C)[6]. It allows peer-to-peer real-time communication between browsers, e.g. video chat and file sharing. When fully implemented in browsers, the applications using WebRTC do not need external plugins, making them very user friendly. There are several benefits to a peer-to-peer model. The path between the peers is shorter, so the delay is lower, which is useful for time sensitive applications such as voice or video chat. Applications can be more distributed since less information needs to go through a central server, and the servers need less processing power.

2.2 Browser compatibility

WebRTC is a relatively new technology. Therefore it is not yet fully implemented in any browsers. However some browsers have implemented enough of the features of WebRTC to be usable. This section will investigate which browsers have working implementations of WebRTC. The browsers use JavaScript, an implementation of ECMAScript, as the programming language for the WebRTC API.



	Canary	Chrome	Opera	Nightly	Firefox	IE	Safari
getUserMedia	🟩	🟩	🟩	🟩	🟩	🟨	🟥
dataChannels	🟩	🟩	🟩	🟩	🟩	🟥	🟥
PeerConnection API	🟩	🟩	🟩	🟩	🟩	🟥	🟥
TURN support	🟩	🟩	🟩	🟩	🟩	🟥	🟥
WebAudio Integration	🟩	🟩	🟩	🟩	🟩	🟥	🟥
MediaStream API	🟩	🟩	🟩	🟨	🟨	🟥	🟥
Echo cancellation	🟩	🟩	🟨	🟨	🟨	🟥	🟥
Solid interoperability	🟨	🟨	🟨	🟨	🟨	🟥	🟥
mediaConstraints	🟨	🟨	🟨	🟨	🟥	🟥	🟥
Multiple Streams	🟩	🟩	🟩	🟥	🟥	🟥	🟥
Screen Sharing	🟨	🟨	🟥	🟨	🟥	🟥	🟥
Simulcast	🟩	🟨	🟥	🟥	🟥	🟥	🟥
Stream re-broadcasting	🟨	🟨	🟥	🟥	🟥	🟥	🟥
ORTC API	🟥	🟥	🟥	🟥	🟥	🟨	🟥

FIGURE 2.1: Screenshot of isWebRTCReadyYet.com taken on the 3rd of October, 2014. This figure shows the status of the different features of WebRTC in the different browsers

Figure 2.1 shows the different features of WebRTC as well as the status of availability in the different browsers. The color green indicates that it is available, yellow to some degree and red not at all. As the figure shows, only Mozilla Firefox, Google Chrome and Opera support major elements of WebRTC. In this thesis I will only focus on Firefox and Chrome, since they are the most used browsers[7].

2.3 How to get a media stream from the computer

To access media streams it is possible to use `getUserMedia`. It is not a part of the WebRTC specification[8], but it is an important aspect for WebRTC to be useful. This feature allows the browser to get access to any cameras or microphones that are connected to the computer. Then the media streams from the devices can be sent over WebRTC, enabling another browser to receive streams directly from the webcam or microphone. This is useful to enable video and audio communication in the browser.

```
navigator.getUserMedia(
        {
                //Constraints
                video: true,
                audio: true,
        },
        function(stream) {
                //Success callback
                console.log('got stream: ' + stream);
        },
        function(error) {
                //Error callback
                console.error(error);
        }
);
```

Listing 1: Using getUserMedia to access media streams

`getUserMedia` is attached to the global navigator object. The function takes three arguments, the first one is a set of constraints that define what kind of stream you want. The second argument is a callback that takes the stream as an argument. The last argument is an error callback.

2.4 How to use peer connections

RTCPeerConnection is the most basic part of WebRTC. It is the connection that binds the endpoints together[6], and allows the users to send data back and forth, for example a media stream from a microphone.

```
var options = {
        iceServers: []
};

var peerConnection = new RTCPeerConnection(options);

peerConnection.addStream(stream);
peerConnection.createOffer(function(offer) {
        peerConnection.setLocalDescription(offer);
        //send offer to other peer
});
```

Listing 2: Creating a peer connection

The above code illustrates how you create a new RTCPeerConnection. Then you need to add the streams you want to include in the connection. The stream you want can be obtained from `getUserMedia`. You then need to create an offer. This offer is specified using Session Description Protocol (SDP)[9]. This Session Description then needs to be sent to the peer you want to connect to.

The following code shows how to use the offer that you receive from the calling peer.

```
var peerConnection = new RTCPeerConnection(options);

peerConnection.setRemoteDescription(
        new RTCSessionDescription(offer),
        function() {
                //Success callback
        },
        function(err) {
                //Error callback
                console.log(err);
        }
);

peerConnection.addStream(stream);
peerConnection.createAnswer(function(answer) {
        peerConnection.setLocalDescription(answer);
        //send answer back to first peer
});
```

Listing 3: Receiving a WebRTC offer

When you receive a Session Description from the other peer you need to use the `setRemoteDescription` function on the `RTCPeerConnection` object. The rest of the procedure is similar to the first one. You need to include the streams you want in the connection and then send a corresponding answer back to the first peer. When the first peer receives the answer it needs to use the `setRemoteDescription` function as well.

2.5 Using data channels

The `RTCDataChannel` interface represents a bi-directional data channel between two peers[6]. It is useful for sending any data that is not in the form of media streams, e.g. text messages or files.

The following code illustrates how one creates a data channel and then sends an offer to another peer.

```
var dataChannel = peerConnection.createDataChannel("dataChannel", {
        ordered: false
});

dataChannel.onopen = function() {
        dataChannel.send('hi');
}

//create and send offer
```

Listing 4: Creating a data channel

The `createDataChannel` function takes two arguments. The first one is the label for the channel, the second one is an object with one or more options. The most noteworthy of these options is the ordered option. This is either true or false. If it is true then the data received on the channel is guaranteed to be in the same order that the data was sent in.

The following snippet shows how the answerer sets up the data channel. The channel will automatically be opened on the answerer side.

```
//receive offer and create RTCPeerConnection

peerConnection.ondatachannel = function(evt) {
        var dataChannel = evt.channel;

        dataChannel.onmessage = function(evt) {
                console.log('Incoming message: ' + evt.data);
        }
}

//create and send answer
```

Listing 5: Receiving a data channel

The RTCDataChannel API supports sending of Strings and also binary data in the form of Blob, ArrayBuffer or ArrayBufferView.

2.6 Screen sharing

Screen sharing allows a video stream of the screen of the desktop to be sent over WebRTC. The feature is not a part of the WebRTC API[6]. It is a feature the browser may choose to include. The browser provides access to a media stream of the screen, and this media stream can be sent over WebRTC in the same way a stream from a camera can be sent. A more in depth look at screen sharing is found in section 5.1.

2.7 WebRTC and the use of bandwidth

When starting a WebRTC streaming connection, the bandwidth usage is very low, less than 500 Kbit/s[10]. This is regardless of the resolution of the video source. This is achieved by using very high levels of compressions and few frames per second. The stream will then adapt to the available bandwidth by increasing the quality of the stream. If WebRTC detects high levels of dropped packages, it will then reduce the bandwidth.

WebRTC does not guarantee the quality of the video. If you send a High Definition stream, WebRTC might and probably will compress it to fit in the available bandwidth. If the bandwidth is sufficient, the stream will eventually reach max quality. If you select a lower quality video, it will require less bandwidth and WebRTC will never use more bandwidth than the upper limit of the stream. Therefore it is possible to use the quality of the video as a way to regulate the use of bandwidth.

WebRTC uses Secure Real-time Transport Protocol (SRTP)[11] and Secure Real-time Control Transport Protocol (SRTCP)[11]. SRTP is used to transmit the media streams. SRTCP provides many useful statistics from the transmission, e.g. jitter, delay and dropped packages. This information is then used by WebRTC to adapt the media streams to the current environment.

2.8 Function prefaces in the browsers

Several of the functions described earlier are prefaced with `webkit` in Google Chrome and `moz` in Mozilla Firefox. To make this easier Google provides a JavaScript tool called `adapter.js`[12]. This tool allows you to use the function names described in the WebRTC standard[6]. For example `navigator.getUserMedia` as opposed to `navigator.webkitGetUserMedia`. It also makes it easy to create WebRTC sessions between Firefox and Chrome, since you don't have to worry about the different browser implementations. For the rest of the thesis, the use of this or a similar tool is implicit.

3. The prototype system

This chapter will describe the existing prototype system created in a previous master thesis[4]. Section 3.1 describes the architecture of the prototype system. Section 3.2 is a general description of the prototype and how to use it. Finally the technologies used for the prototype that are relevant for this thesis will be covered in section 3.3.

3.1 Architecture

The prototype system uses a centralized server structure. All communications between two clients pass through a server. There are three separate servers. The application server is the server that serves the web application, it also does all the signalling on behalf of the web clients. Then there is the Dialogic XMS[13] server. It acts as a media bridge, and all the video and audio streams go through this server. It takes care of the different SDP formats, and enables WebRTC clients and Session Initiation Protocol (SIP) clients to communicate seamlessly. If you are in a conference with several other peers, you upload your own stream to the server, and you download one stream. All the streams from the different peers are multiplexed into one. This has several benefits, it saves bandwidth and it saves on processing power on the client side. Although it requires more resources from the server, since all the video streams are processed in real time. Finally there is the Gintel Multimedia Private Branch Exchange (MPBX) platform. This is responsible for bridging the gap between the general telephony network and the application network, making it possible to use a normal cell phone to call into a conference.

This thesis will focus on the WebRTC side of the network, and not the SIP side. Figure 3.2 depicts this part of the prototype network. The features described in chapter 5 are designed with WebRTC peer-to-peer communication in mind.

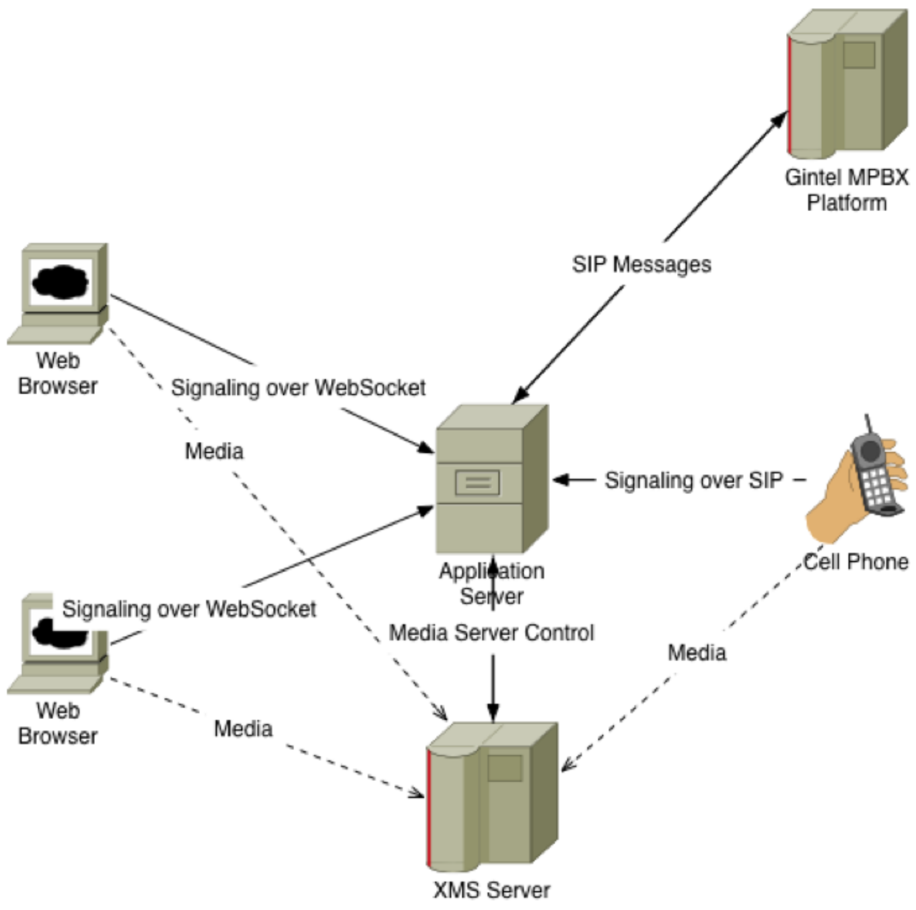


FIGURE 3.1: Figure[4] showing the different components in the prototype system

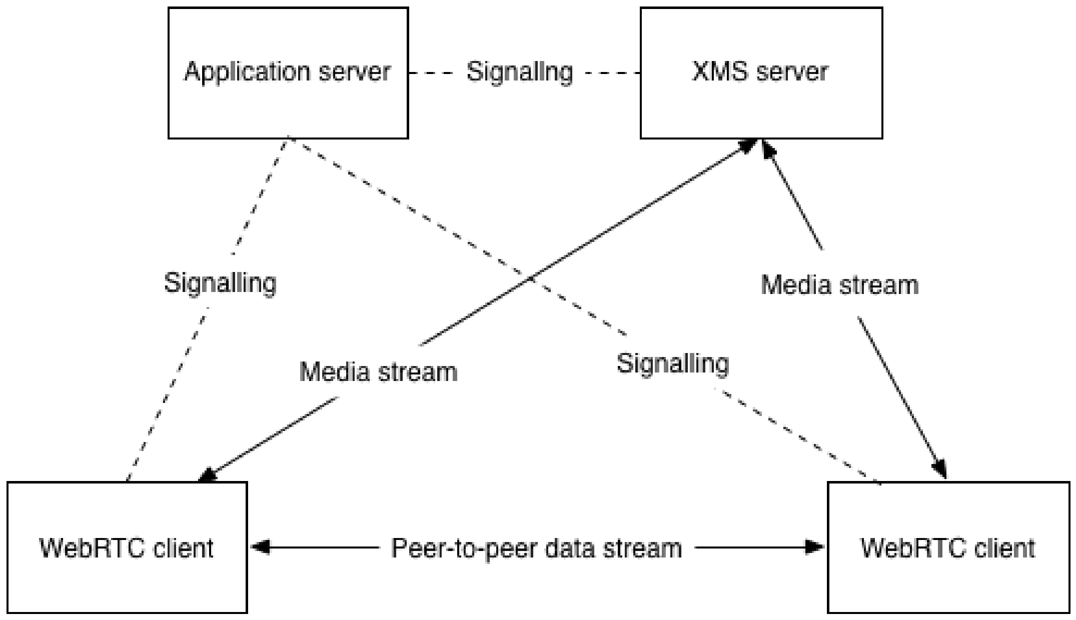
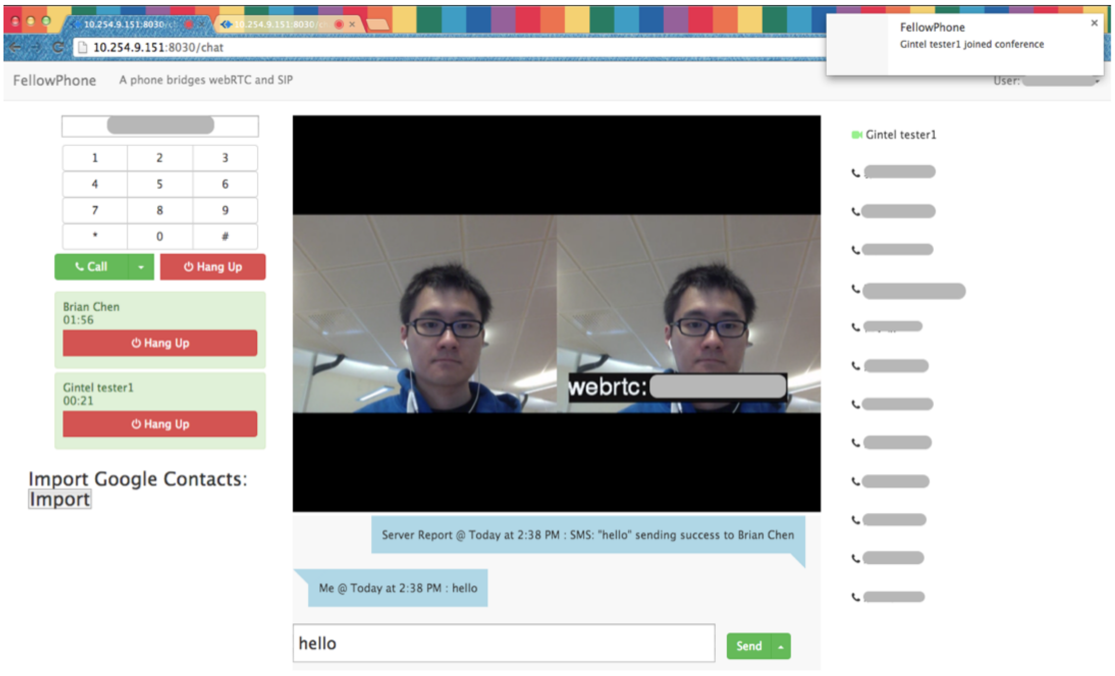


FIGURE 3.2: Figure showing a subset of the network with peer-to-peer communication between the clients

3.2 Description

When you open the prototype application you are greeted with a login screen. Here you login with a telephone number, and a password. To login to the application a user must be created and authorized beforehand. It is possible to call and send text messages to other phone numbers, and the owner of the phone number logged in will be billed for these services. To call another client, you enter a phone number. If that number is logged into the application the call will be a WebRTC based call. If it is a phone on the telephony network, the XMS server will translate from WebRTC to SIP. When a call is in progress it is a working conference call. Therefore it is possible to call other clients during a call, and if they accept they will be included in the call. The features implemented include instant messaging, using both SMS and direct Internet messaging, video and audio conference and file sharing.



FellowPhone
Gintel tester1 joined conference

Server Report @ Today at 2:38 PM : SMS: "hello" sending success to Brian Chen

Me @ Today at 2:38 PM : hello

hello

FIGURE 3.3: [4] Figure showing a conference between three clients, two are participating using the web client and one client is connected with a phone. The two other clients are shown on the left side with their own hangup button. It is also possible to send messages using the chat field

3.3 Relevant technology

This section will go through some of the technologies used by the prototype system that are relevant to this thesis.

3.3.1 The NodeJS server

NodeJS[14] is the server used for serving the application, and communicating with the XMS server and the SIP system. NodeJS is used for the messaging and file sharing between the different clients. This means that the files sent between the clients go through the server, and the transfer is not peer-to-peer. This is not optimal because it puts strain on the server, and increases the time it takes to transmit a file.

3.3.2 The AngularJS framework

The prototype system uses AngularJS[15] for the client. AngularJS is a JavaScript framework that makes it possible to create single-page applications[16]. The normal way to create web pages is to create different HTML files for each page, and use hyperlinks to navigate between them. When a link is clicked by a user, the browser will then download the new document from the server and display it to the user. In a single-page application, it is only one web page, but the content is changed dynamically using JavaScript. This has the benefit of being much more responsive, because it is not necessary to download new files from the server every time one wishes to see new content. All the content is contained in one page, this means that initially it will take longer to download the application, but it is a one time download. It is also possible to dynamically download the resources when needed, allowing for the application to be loaded faster.

AngularJS offers several useful tools, and I will describe the ones that are relevant to my thesis.

3.3.2.1 Module

A module[17] is a container for different AngularJS components. You can divide your application into different modules to make the parts more independent of each other. Using a module makes it easy to include your tools into other projects.

```
var app = angular.module('testModule', []);
```

Listing 6: Creating an AngularJS module

This is how you create an AngularJS module. The object `app` can now be used to add other components or configure you module. How to include another module in your project is shown below.

```
var app = angular.module('testModule', ['anotherModule']);
```

Listing 7: Including another module

When you include another module, all of the factories and directives from that module becomes available in your project.

3.3.2.2 Factory

A factory[18] is an AngularJS component that is used to perform most operations. If you need a component to perform some tasks that are not related to the GUI then a factory is useful. The following example shows a basic usage of the AngularJS factory.

```
app.factory('calculator', function() {
        var calculator = {
                add: function(a, b) {
                        return a + b;
                }
        };

        return calculator;
});
```

Listing 8: Creating a simple calculator factory

This example shows how to create a basic calculator factory that is capable of adding two numbers.

```
app.run( function (calculator) {
        console.log(calculator.add(1, 3));
});
```

Listing 9: Using the calculator factory

To use the factory you must first inject[15] it. When you provide the `calculator` as an argument to one of the AngularJS components, the factory will be made available.

3.3.2.3 Directive

A directive[19] is a component that can interact with the Document Object Model (DOM). A principle of AngularJS is to keep most of the logic of the application separated from the GUI, and use directives when you wish to alter the GUI.

```
<div ng-show="true">Hello
</div>
```

Listing 10: Using a predefined directive

This simple example shows the use of the directive `ng-show`. The directive takes a boolean argument, and shows or hides the element according to the value.

4. Use cases

Developing use cases are a great way to gain insight into what an application will be used for, and what features are useful. In this chapter I will describe three different potential scenarios using the application. I will use these scenarios to justify what features I will develop or improve for the prototype. In chapter 5 the improvements are chosen and described in detail. The use cases investigated were chosen based on plausible uses for the application. While being completely separate, they span a large portion of potential implementations when combined.

4.1 Online education

A teacher uses the application, to share video streams, slides and files. The students can participate to different degrees, either completely passively or using voice, video and/or text chat to ask questions or comment.

Stakeholders: Institution/teacher, student, content.

Technologies: voice call, video call, screen sharing, file sharing, whiteboard.

4.1.1 Live lecture

A professor plans to have a lecture. He creates a lecture instance using the application, and receives a link to a specific web page. He shares this link using the online school bulletin board, and states that the lecture will begin the day after at 13 pm. This lecture will be held in an auditorium with an audience, but it will also be available online in real time.

The day after, the professor logs in to his account using the application from his office, 30 minutes before the lecture starts. He checks that he has all the slides and files needed. He then walks to the auditorium to start the lecture. At 13 pm, he starts the online lecture by enabling it in the application. Then everyone with a link to the lecture can see what the professor shares with them. The professor has enabled the lecture slides to be shown, as well as live video from the lecture. He has also uploaded several files that are relevant to the lecture, and the students participating in the online lecture can download the files. 30 students are in the auditorium when the lecture starts, and 13 students have connected using the application. When students in the auditorium have questions the students using the online application can hear what they say. The students that are participating online can also ask questions, either by using their microphone or camera, or write the question using text. The students in the auditorium will hear the question if it is streamed using audio. If the question is sent using text, the question will show up on the application the professor uses. He can then read the question out loud, and answer it.

This approach needs two different clients to the application. One teacher client and one student client. The teacher client can create lectures and control them, while the student client can watch and participate.

4.1.2 Live streaming of prerecorded lecture

The day after the live lecture the same professor wants the students that missed the lecture the day before to have another chance to catch it. He sets up a repeat lecture using the saved video from the lecture the day before, and the previous lecture is streamed to the online students in much the same way as the day before, still with slides and files. The students will still be able to ask questions using text chat while the lecture is playing. This time however the professor has other things he needs to do, therefore a student with a firm grasp on the curriculum chosen by the professor will answer any questions the other students might have.

4.1.3 Archive of prerecorded lectures

All the prerecorded lectures can also be accessed by the students at any time. This is strictly one way, and it is not possible to ask questions because there is no other participants in this lecture.

4.2 Health care provider

In this use case it is imagined that the application can be used for emergency services e.g calling to 911 with a health issue.

Stakeholders: Hospital, caller.

Technologies: Voice call, video call.

A woman is having a semi severe medical emergency. She does not know what the problem is, but her daughter calls 911. A worker for the emergency central answers the call. The daughter says she needs an ambulance, and the worker dispatches one right away. In the mean time the worker asks the daughter to describe the symptoms. The daughter struggles to accurately portray the symptoms. The worker then asks if the daughter has a computer nearby or a smartphone. The daughter replies that she is using a smartphone right now. The emergency worker sends a link to the number the daughter is calling from using SMS. The daughter opens this link on the smartphone using a web browser. Now the daughter can use the attached camera on the phone to show the medical worker the symptoms of the mother, and the medical worker can provide sound advice while they are waiting for the ambulance.

4.3 Real estate agent

Stakeholders: Agent and customer.

Technologies: Voice/video call, screen sharing, file sharing, whiteboard.

A man is interested in buying a house, and calls up his realtor. The man explains what kind of house he wants and the realtor says that he thinks he knows exactly what the man wants. The agent asks if the man is near a computer with internet access, the man confirms this. The realtor then sends a link to the man using email. The link takes the man to the application where he can connect with his webcam and have a live conversation with the realtor. The realtor has several possible homes for the man to look at, and he displays them on the mans screen using the application. The man can see pictures of the house, the man can also download a complete document with all the information and pictures of the different houses. The realtor explains the different features of the house while he shows the man corresponding pictures and information. He also opens up a map where the realtor draws circles around the positions of the houses.

This application needs two different clients. One client for the realtor where he can control what the customer sees, and one client for the customer which passively observes the information the realtor chooses to share with the customer.

5. Improving the prototype

The main focus of this thesis was to research and experiment with WebRTC technology by using the prototype described in chapter 3 as a foundation. Improving the existing prototype has several benefits. The obvious one is that it will improve, secondly it is a great way to discover and understand WebRTC features and applications. By choosing the features to include, then investigating how to actually implement and use them, a lot can be uncovered of the workings of WebRTC.

This chapter will describe the development of several improvements for the prototype, and the rationale behind the design choices. The improvements have been chosen based on the use cases described in chapter 4. There are three improvements detailed in the chapter, screen sharing in section 5.1, whiteboard in section 5.2 and file sharing in section 5.3. The method behind the choices is described in section 1.1.2.

Because of unforeseen problems the prototype system was not ready to be extended at the time of this thesis. This is explained further in section 6.1.2. Therefore a major design goal of the design and implementation became apparent, namely modularity. It was important for the features to be self-contained and easily installed. The improvements were still designed and created as planned, but the final step of integrating them in the prototype was omitted.

5.1 Screen sharing

Screen sharing can be useful in several of the use cases described in chapter 4. In use case 4.1, online education, the professor can use screen sharing to show relevant things to the current lecture. In use case 4.3, realtor-customer, screen sharing can be used to show pictures of houses or point to a map.

This section will investigate how to use screen sharing in modern browsers.

5.1.1 Enable TLS

Using `getUserMedia` to capture the screen is only possible while using Transport Layer Security (TLS) at this time. The first step is to setup the prototype server to use HTTPS.

```
var https = require('https');
var app = require('express')();
var fileSystem = require('fs');

var options = {
        key: fileSystem.readFileSync('server.key'),
        cert: fileSystem.readFileSync('server.crt')
};

https.createServer(options, app).listen(443);
```

Listing 11: How to enable TLS in NodeJS

To use a secure server you need a public key certificate[20] and a private key.

5.1.2 Google Chrome

One of the browsers that have implemented support for screen sharing is Google Chrome. This section will describe the different ways screen capture can be achieved.

5.1.2.1 Command line flag

Chrome 35 has a command line flag that can be used to get a media stream from the screen[21]. `--enable-usermedia-screen-capturing`. By opening Google Chrome with this flag, screen capturing will be available. It is then possible to access screen capture with `getUserMedia` using the constraints `chromeMediaSource: 'screen'`.

```
navigator.getUserMedia(
        {
                //Constraints
                video: {
                        mandatory: {
                                chromeMediaSource: 'screen'
                        }
                },
        },
        function(stream) {},
        function(error) {}
);
```

Listing 12: Getting a media stream from the screen using the command line flag

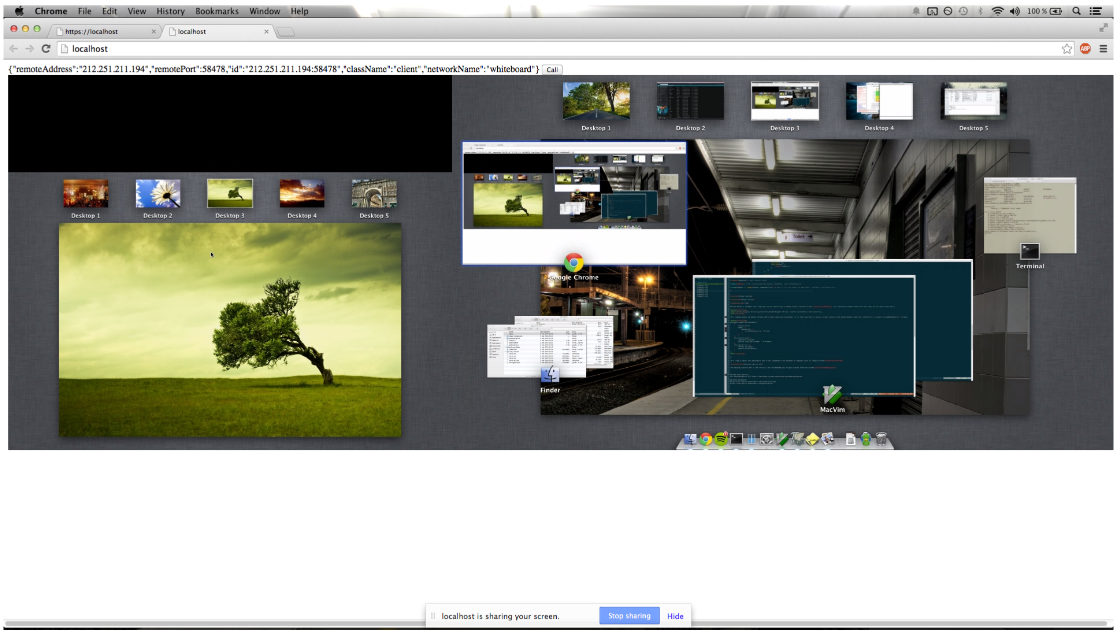

```
{"remoteAddress":"212.251.211.194","remotePort":58478,"id":"212.251.211.194:58478","className":"client","networkName":"whiteboard"}
```

FIGURE 5.1: Figure showing the media stream obtained using the –enable-usermedia-screen-capturing flag

Figure 5.1 shows the flag in use. The figure is a screenshot of the media stream obtained by using the flag. The desktop depicted on the screenshot used two computer monitors. The stream shows both of these screens next to each other in the same frame. The screen on the left has 800 vertical lines of resolution, the one on the right has 1080 lines. This results in the black bar to the top left, which is empty space not being displayed by any screen.

This flag is meant for developers and is not supposed to be exposed to regular users of Google Chrome[21].

5.1.2.2 Creating an extension for Google Chrome

The Desktop Capture API is the official and recommended way to get streams from the screen[22]. It is only available to Google Chrome extensions. I created a simple extension that uses this API, and will describe it next.

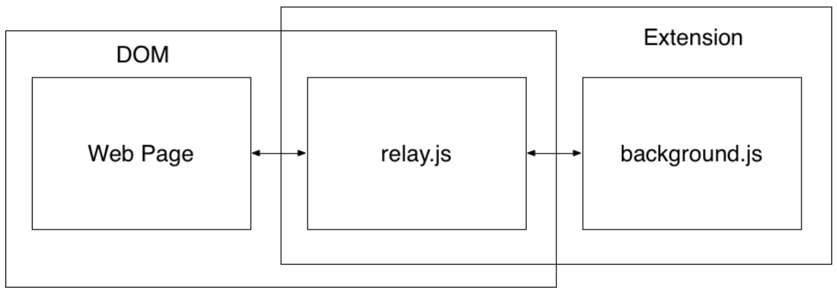


FIGURE 5.2: This figure shows the different files of the extension and how they interact with the web page

`background.js`, is an event page[23]. This is a JavaScript file that runs in the background and is always available when needed. Only one instance of the file exists at any one point. This is the file that is the most important since it is the one that utilizes the Desktop Capture API. This is shown in listing 13.

```
chrome.desktopCapture.chooseDesktopMedia(
        ["screen", "window"],
        sender.tab,
        function(id) {
                //id is the value that is needed to get the stream.
                //send this id back to the web page
        }
);
```

LISTING 13: How the Desktop Capture API is used in `background.js`

The `chrome.desktopCapture.chooseDesktopMedia` function call is the one that uses the API. This function displays a dialog where the user can select the stream they want. It takes three arguments. The first one is an array of `DesktopCaptureSourceType`, it defines the different types of sources the user gets to choose between. The second argument is optional but required if the extension itself is not the one that requests the stream. It is the target tab that wants to use the stream. The third argument is the callback which is called with the media id of the source. This id is the important part and can be used in combination with `getUserMedia` to access a stream of the desktop.

`relay.js` is a content script[24]. It is responsible for communication between the web page that wants to use the extension and the actual extension. Content scripts are injected into web pages where they have access to the DOM. The DOM is the structure of elements on the page, therefore a content script can change the appearance of a web page. However it cannot directly use other JavaScript code running on the same web page. It is isolated, and the functions and variables of the content script is not accessible from the rest of the web page. It is still possible to communicate with the web page using the DOM. This is achieved using message passing[25]. By adding a message listener to the global `window`, it is possible to send messages to and from the extension with `window.postMessage`. Figure 5.3 shows how the messages are passed around the extension, and how the media source id ends up in the web page.

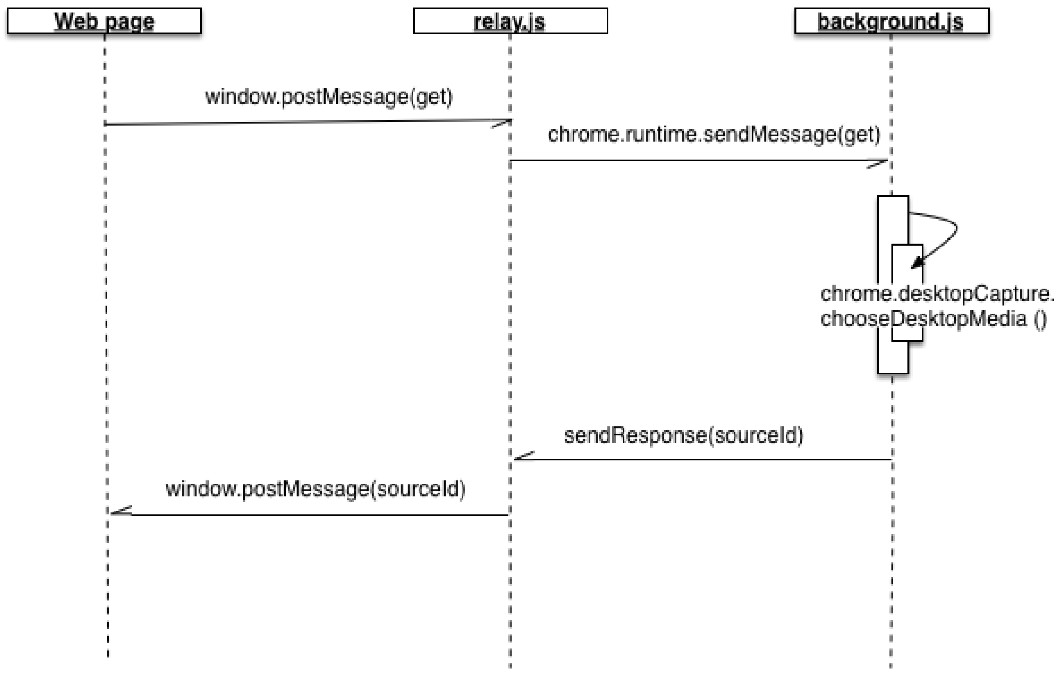


FIGURE 5.3: Sequence diagram of how messages are passed along inside the extension

If you want to use the extension in your web page you can do so by using `window.postMessage`. The following is an example illustrating how the extension can be used.

```
window.addEventListener('message', function(event) {
        var sourceId = event.data.screenCapture.id;
        console.log(sourceId);
});

var message = {
        screenCapture: {
                type: 'get',
        }
};

window.postMessage(message, '*');
```

Listing 14: How to use the extension

This is enough to get the media source id. You need an event listener to receive messages from the extension, and you need to send messages when you want to interact with it. This code will tell the extension to show the media picker dialog shown in figure 5.4. When you select a media source, you receive a source id. Listing 15 shows how the source id can then be used to access the stream.

```
navigator.getUserMedia(
        {
                //Constraints
                video: {
                        mandatory: {
                                chromeMediaSource: "desktop",
                                chromeMediaSourceId: sourceId,
                        },
                }
        },
        function(stream) {},
        function(error) {}
);
```

Listing 15: Using the source id obtained from the extension to get the stream

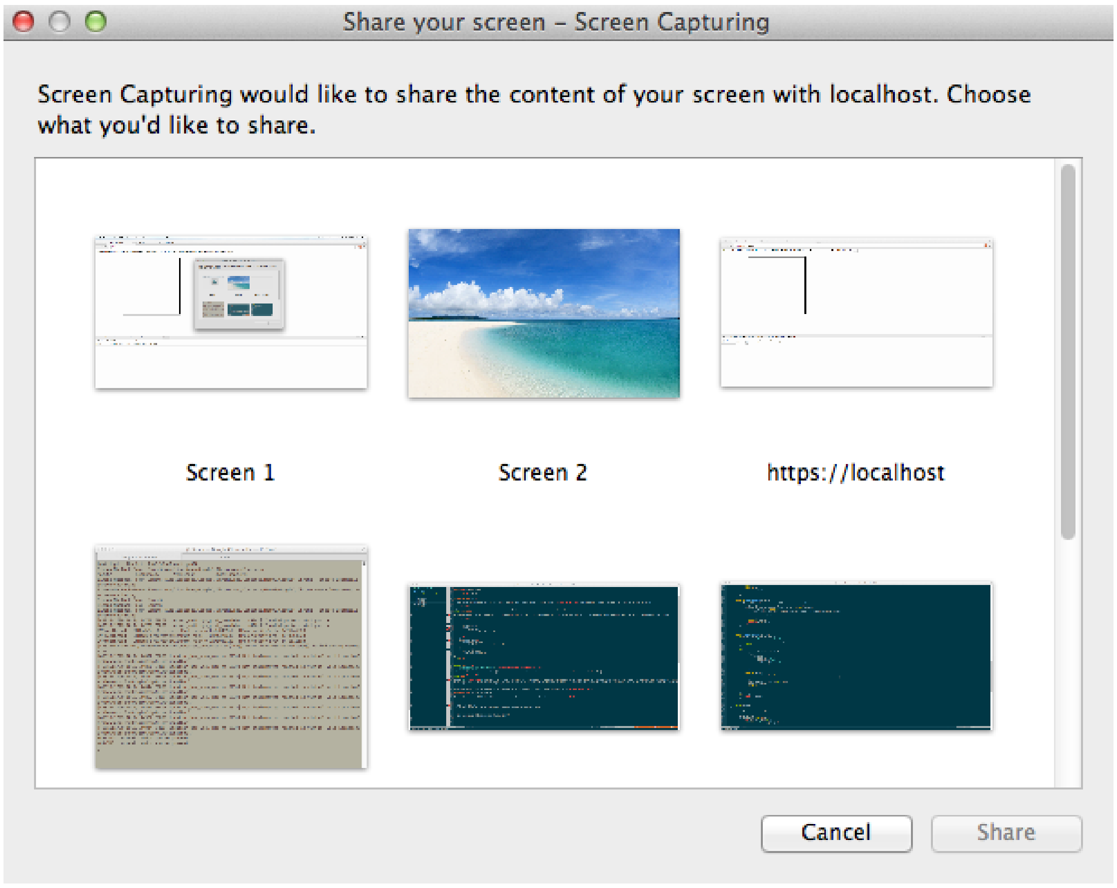

Share your screen – Screen Capturing
Screen Capturing would like to share the content of your screen with localhost. Choose what you'd like to share.
Screen 1
Screen 2
https://localhost
Cancel
Share

FIGURE 5.4: Screenshot showing the screen selector that enables the user to select specific applications or screens

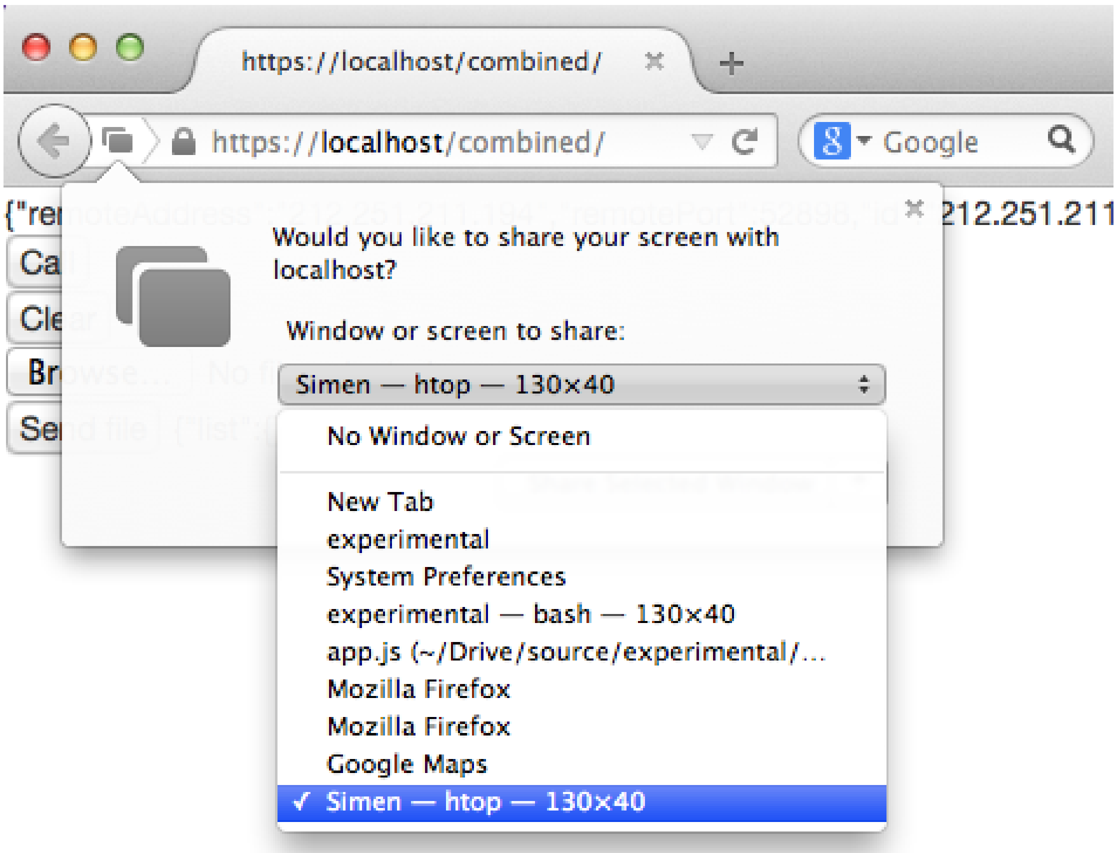
The next step is to install the extension in Google Chrome. There are two ways to accomplish this. It is possible to load an unpackaged extension if developer mode is enabled in the browser. However the most natural way to install the extension is to download the packaged extension. This is a file with a .crx filetype. Then open up the extension page of Google Chrome and simply drag the file over the window and drop it onto the browser window. Then accept the permissions the extension needs.

5.1.3 Mozilla Firefox

Firefox also supports screen capture. But it requires a little configuration. In Firefox version 34 you can open `about:config` from the address bar, then you get access to a lot of settings. Then you need to find `media.getusermedia.screensharing.allowed_domains`. This is a list of domains that are allowed to use the screen capture feature. Then just add your domain to the list. Then any application from that domain can use this feature. Listing 16 shows how to get the stream using `getUserMedia`.

```
navigator.getUserMedia(
        {
                //Constraints
                video: {
                        mozMediaSource = 'window';
                        mediaSource = 'window';
                },
        },
        function(stream) {},
        function(error) {}
);
```

LISTING 16: Using screen capture in Firefox

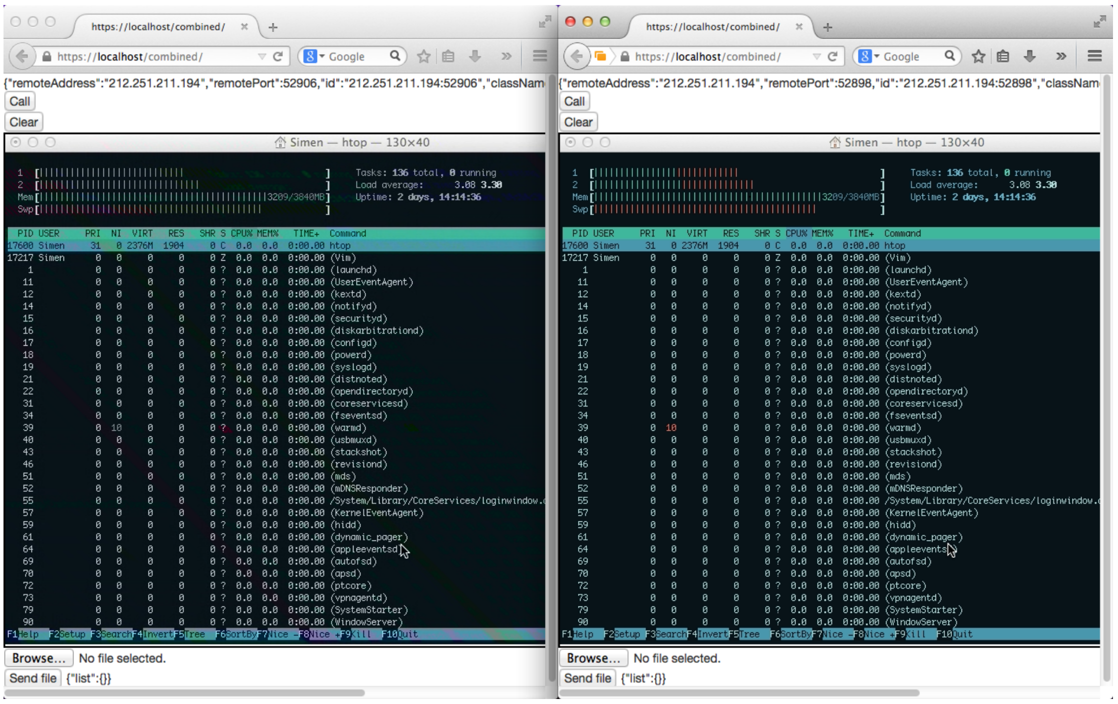


https://localhost/combined/

Would you like to share your screen with localhost?

Window or screen to share: Simen — htop — 130×40

FIGURE 5.5: This figure shows a screenshot of the dialog that allows you to select a media source.



https://localhost/combined/

{"remoteAddress":"212.251.211.194","remotePort":52906,"id":"212.251.211.194:52906","classNam

```
  1  [|||||||||||||||||||||||||||                        ]   Tasks: 136 total, 0 running
  2  [|||||||||||||||||||||||||||||                      ]   Load average:      3.08 3.30
  Mem[|||||||||||||||||||||||||||||||||||||||||3209/3840MB]   Uptime: 2 days, 14:14:36
  Swp[||||||||||||||||||||||||||||||||||||||||||          ]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
17600 Simen      31   0 2376M  1904     0 C  0.0  0.0  0:00.00 htop
17217 Simen       0   0     0     0     0 Z  0.0  0.0  0:00.00 (Vim)
    1             0   0     0     0     0 ?  0.0  0.0  0:00.00 (launchd)
   11             0   0     0     0     0 ?  0.0  0.0  0:00.00 (UserEventAgent)
   12             0   0     0     0     0 ?  0.0  0.0  0:00.00 (kextd)
   14             0   0     0     0     0 ?  0.0  0.0  0:00.00 (notifyd)
   15             0   0     0     0     0 ?  0.0  0.0  0:00.00 (securityd)
   16             0   0     0     0     0 ?  0.0  0.0  0:00.00 (diskarbitrationd)
   17             0   0     0     0     0 ?  0.0  0.0  0:00.00 (configd)
   18             0   0     0     0     0 ?  0.0  0.0  0:00.00 (powerd)
   19             0   0     0     0     0 ?  0.0  0.0  0:00.00 (syslogd)
   21             0   0     0     0     0 ?  0.0  0.0  0:00.00 (distnoted)
   22             0   0     0     0     0 ?  0.0  0.0  0:00.00 (opendirectoryd)
   31             0   0     0     0     0 ?  0.0  0.0  0:00.00 (coreservicesd)
   34             0   0     0     0     0 ?  0.0  0.0  0:00.00 (fseventsd)
   39             0  10     0     0     0 ?  0.0  0.0  0:00.00 (warmd)
   40             0   0     0     0     0 ?  0.0  0.0  0:00.00 (usbmuxd)
   43             0   0     0     0     0 ?  0.0  0.0  0:00.00 (stackshot)
   46             0   0     0     0     0 ?  0.0  0.0  0:00.00 (revisiond)
   51             0   0     0     0     0 ?  0.0  0.0  0:00.00 (mds)
   52             0   0     0     0     0 ?  0.0  0.0  0:00.00 (mDNSResponder)
   55             0   0     0     0     0 ?  0.0  0.0  0:00.00 /System/Library/CoreServices/loginwindow.
   57             0   0     0     0     0 ?  0.0  0.0  0:00.00 (KernelEventAgent)
   59             0   0     0     0     0 ?  0.0  0.0  0:00.00 (hidd)
   61             0   0     0     0     0 ?  0.0  0.0  0:00.00 (dynamic_pager)
   64             0   0     0     0     0 ?  0.0  0.0  0:00.00 (appleeventsd)
   69             0   0     0     0     0 ?  0.0  0.0  0:00.00 (autofsd)
   70             0   0     0     0     0 ?  0.0  0.0  0:00.00 (apsd)
   72             0   0     0     0     0 ?  0.0  0.0  0:00.00 (ptcore)
   73             0   0     0     0     0 ?  0.0  0.0  0:00.00 (vpnagentd)
   79             0   0     0     0     0 ?  0.0  0.0  0:00.00 (SystemStarter)
   90             0   0     0     0     0 ?  0.0  0.0  0:00.00 (WindowServer)
F1Help  F2Setup F3SearchF4InvertF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

No file selected.

{"list":{}}

https://localhost/combined/

{"remoteAddress":"212.251.211.194","remotePort":52898,"id":"212.251.211.194:52898","classNam

```
  1  [|||||||||||||||||||||||||||                        ]   Tasks: 136 total, 0 running
  2  [|||||||||||||||||||||||||||||                      ]   Load average:      3.08 3.30
  Mem[|||||||||||||||||||||||||||||||||||||||||3209/3840MB]   Uptime: 2 days, 14:14:36
  Swp[||||||||||||||||||||||||||||||||||||||||||          ]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
17600 Simen      31   0 2376M  1904     0 C  0.0  0.0  0:00.00 htop
17217 Simen       0   0     0     0     0 Z  0.0  0.0  0:00.00 (Vim)
    1             0   0     0     0     0 ?  0.0  0.0  0:00.00 (launchd)
   11             0   0     0     0     0 ?  0.0  0.0  0:00.00 (UserEventAgent)
   12             0   0     0     0     0 ?  0.0  0.0  0:00.00 (kextd)
   14             0   0     0     0     0 ?  0.0  0.0  0:00.00 (notifyd)
   15             0   0     0     0     0 ?  0.0  0.0  0:00.00 (securityd)
   16             0   0     0     0     0 ?  0.0  0.0  0:00.00 (diskarbitrationd)
   17             0   0     0     0     0 ?  0.0  0.0  0:00.00 (configd)
   18             0   0     0     0     0 ?  0.0  0.0  0:00.00 (powerd)
   19             0   0     0     0     0 ?  0.0  0.0  0:00.00 (syslogd)
   21             0   0     0     0     0 ?  0.0  0.0  0:00.00 (distnoted)
   22             0   0     0     0     0 ?  0.0  0.0  0:00.00 (opendirectoryd)
   31             0   0     0     0     0 ?  0.0  0.0  0:00.00 (coreservicesd)
   34             0   0     0     0     0 ?  0.0  0.0  0:00.00 (fseventsd)
   39             0  10     0     0     0 ?  0.0  0.0  0:00.00 (warmd)
   40             0   0     0     0     0 ?  0.0  0.0  0:00.00 (usbmuxd)
   43             0   0     0     0     0 ?  0.0  0.0  0:00.00 (stackshot)
   46             0   0     0     0     0 ?  0.0  0.0  0:00.00 (revisiond)
   51             0   0     0     0     0 ?  0.0  0.0  0:00.00 (mds)
   52             0   0     0     0     0 ?  0.0  0.0  0:00.00 (mDNSResponder)
   55             0   0     0     0     0 ?  0.0  0.0  0:00.00 /System/Library/CoreServices/loginwindow.
   57             0   0     0     0     0 ?  0.0  0.0  0:00.00 (KernelEventAgent)
   59             0   0     0     0     0 ?  0.0  0.0  0:00.00 (hidd)
   61             0   0     0     0     0 ?  0.0  0.0  0:00.00 (dynamic_pager)
   64             0   0     0     0     0 ?  0.0  0.0  0:00.00 (appleeventsd)
   69             0   0     0     0     0 ?  0.0  0.0  0:00.00 (autofsd)
   70             0   0     0     0     0 ?  0.0  0.0  0:00.00 (apsd)
   72             0   0     0     0     0 ?  0.0  0.0  0:00.00 (ptcore)
   73             0   0     0     0     0 ?  0.0  0.0  0:00.00 (vpnagentd)
   79             0   0     0     0     0 ?  0.0  0.0  0:00.00 (SystemStarter)
   90             0   0     0     0     0 ?  0.0  0.0  0:00.00 (WindowServer)
F1Help  F2Setup F3SearchF4InvertF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

No file selected.

{"list":{}}

FIGURE 5.6: Screenshot of a screen sharing session in Firefox. The window on the left is sharing a separate terminal window

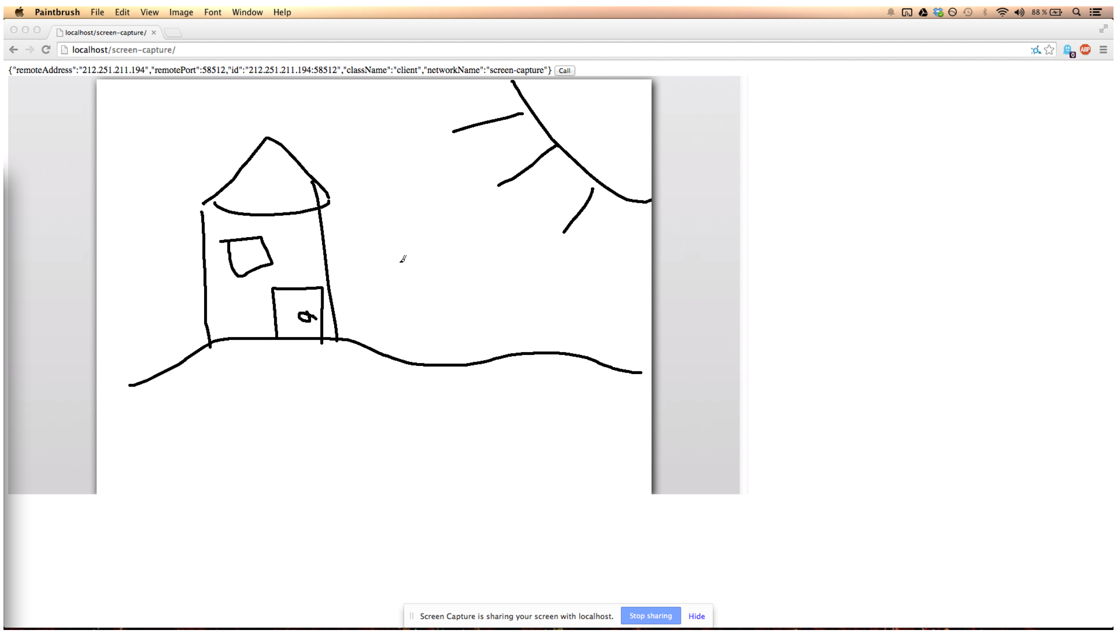
5.2 Whiteboard

A whiteboard is another feature mentioned in the use cases in chapter 4. In use case 4.1, online education, a whiteboard can be used to draw equations and figures to show the students. In use case 4.3, realtor-customer, a whiteboard can be used to emphasize locations on a map.

This section will investigate how a whiteboard feature can be designed and implemented.

5.2.1 Using screen capture

Using what was discovered in section 5.1, it is possible to implement a simple whiteboard using screen sharing. This is accomplished by having an open window where the user can draw. This can be a window attached to the web page or a separate program e.g. Microsoft Paint.



```
{"remoteAddress":"212.251.211.194","remotePort":58512,"id":"212.251.211.194:58512","className":"client","networkName":"screen-capture"}
```

Screen Capture is sharing your screen with localhost.

FIGURE 5.7: Figure showing the media stream from a separate drawing application

Figure 5.7 shows the stream from a drawing application sent over WebRTC. This approach is the most basic and requires no additional implementation in the prototype. All you need is to open the drawing program of your choice and share the screen.

This solution has several drawbacks. It is strictly one way. There is no way for more than one peer to participate. There is also no way to easily switch control of the whiteboard. It would be possible to save the file one peer is drawing, then send it to another peer. The other peer can then share their screen, open the file in a drawing program and continue the drawing. While this is certainly possible, it is not very user friendly.

Another reason whiteboard with screen sharing is not optimal is the fact that the information is sent using a video stream. Sending 30 frames per second of a user drawing can be a lot of wasted bandwidth, especially since a drawing is mostly pixels of the background, or static pixels not changed in the last few seconds.

Whiteboard with screen capture requires no implementation, but offers very little usability and poor performance.

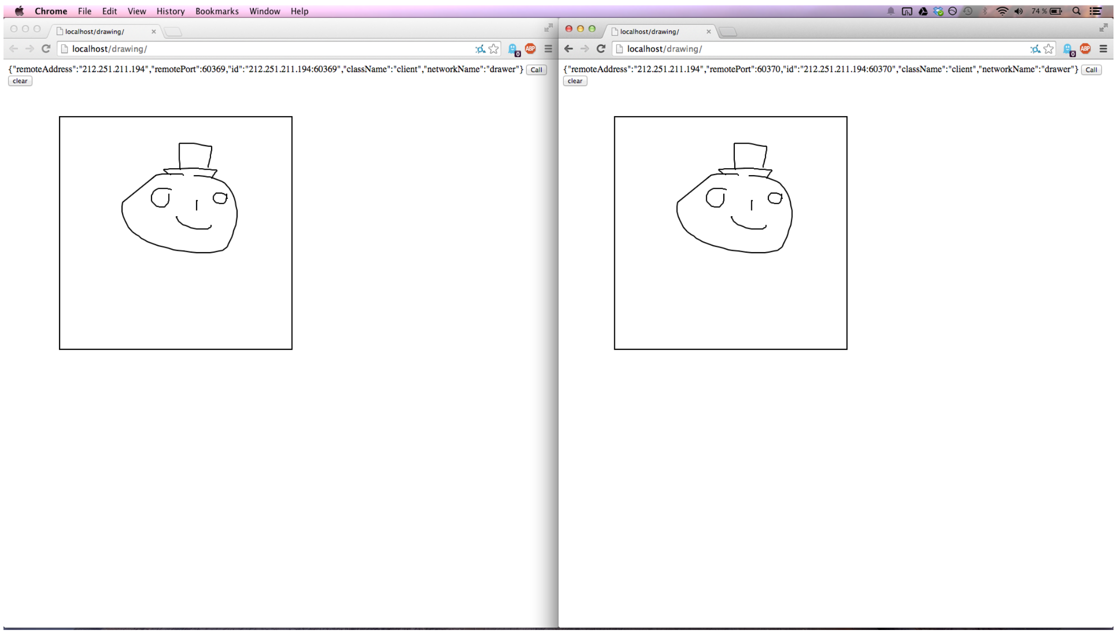
5.2.2 Using data channels

WebRTC offers a way to send data over the connection and that is the use of data channels[6]. This data channel can be used to send real-time information to other users. The information received can then be used to recreate a drawing done by the other user.

HTML5 offers a canvas element that can easily be used to create drawings, and will serve as the basis for the whiteboard implementation. To draw a line you need two points. One start point and a stop point. These points are specified with an x and a y-coordinate. Therefore four values are needed to draw a line. Only two values are needed to create a point. As a user draws on the canvas, the information needed to recreate the drawing can be sent in real time to another user using a data channel. This is the core concept for a whiteboard application. Appendix A shows a prototype implementation of this solution.

This solution is superior to the solution in section 5.2.1 in several ways. It requires a lot less bandwidth since it only sends enough information to recreate the drawing. It is also possible to pass information back and forth between peers using data channels, therefore it is possible for more than one user to use the whiteboard at the same time.

Figure 5.8 is a screenshot of the whiteboard prototype in action. Two tabs are using a shared whiteboard. Both tabs can draw on the canvas using the mouse pointer, and both tabs will see the changes the other tab makes.



{"remoteAddress":"212.251.211.194","remotePort":60369,"id":"212.251.211.194:60369","className":"client","networkName":"drawer"}

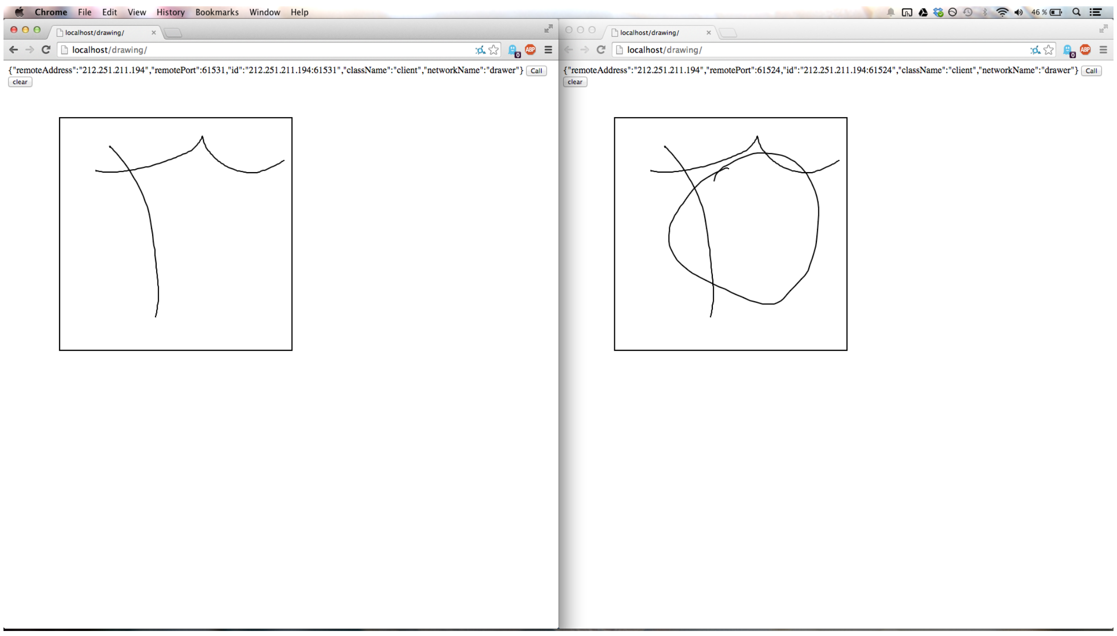
{"remoteAddress":"212.251.211.194","remotePort":60370,"id":"212.251.211.194:60370","className":"client","networkName":"drawer"}

FIGURE 5.8: Screenshot of two tabs in Google Chrome using data channels to enable a two-way whiteboard session

This is a very simple implementation of a whiteboard feature using data channels. It is very basic and could use several additional features such as eraser, different colors etc.

5.2.2.1 Concurrency

When an erase feature is introduced in the whiteboard prototype, concurrency can become a problem. Because it is possible to erase information from the canvas, the order of operations becomes very important. In figure 5.9 the two tabs are experiencing problems with concurrency. They have different versions of the drawings. They both have the exact same information. However the order the canvas operations were performed in were not the same for the two tabs. Therefore they end up with differing drawings.



```
{"remoteAddress":"212.251.211.194","remotePort":61531,"id":"212.251.211.194:61531","className":"client","networkName":"drawer"}
```

```
{"remoteAddress":"212.251.211.194","remotePort":61524,"id":"212.251.211.194:61524","className":"client","networkName":"drawer"}
```

FIGURE 5.9: This screenshot illustrates problems with concurrency that can arise when the order of operations are not consistent

A very simple way to address this is to include a timestamp in the status message. By ordering the operations based on the time they where issued it is possible to remedy some of the situations where concurrency causes problems. However this requires that the peers communicating have synchronized time. Many computers have their clocks set automatically using some time service on the Internet. So it is likely that the clocks of two computers are synchronized, making this approach feasible, but it is not very reliable. If the clocks are not synchronized, the entire application could fall apart, since one user would always have a lower timestamp than the others. The order of operations would still be the same for all users, meaning that technically concurrency would not be a problem. Usability would still suffer if the drawing did not consistently display what you tried to draw, especially if the network had a large delay to deliver the information between the peers. A possible result of this is that users that erase part of the drawing, ends up erasing lines that someone added after the erasing took place. Then the user would erase more that they intended to.

5.2.3 Combining whiteboard and screen sharing

A whiteboard alone can be useful to create drawings and share them, but not all that useful in any of the use cases described. But combining a whiteboard with screen sharing can be useful to underline or emphasize parts of the screen. Figure 5.10 shows how combining whiteboard and screen sharing can be useful in a realtor-customer scenario described in section 4.3. The user on the left is the realtor, using Google Chrome with the screen sharing extension. The user on the right is the customer interested in buying properties. The realtor then draws on the map to show the location of four different houses.

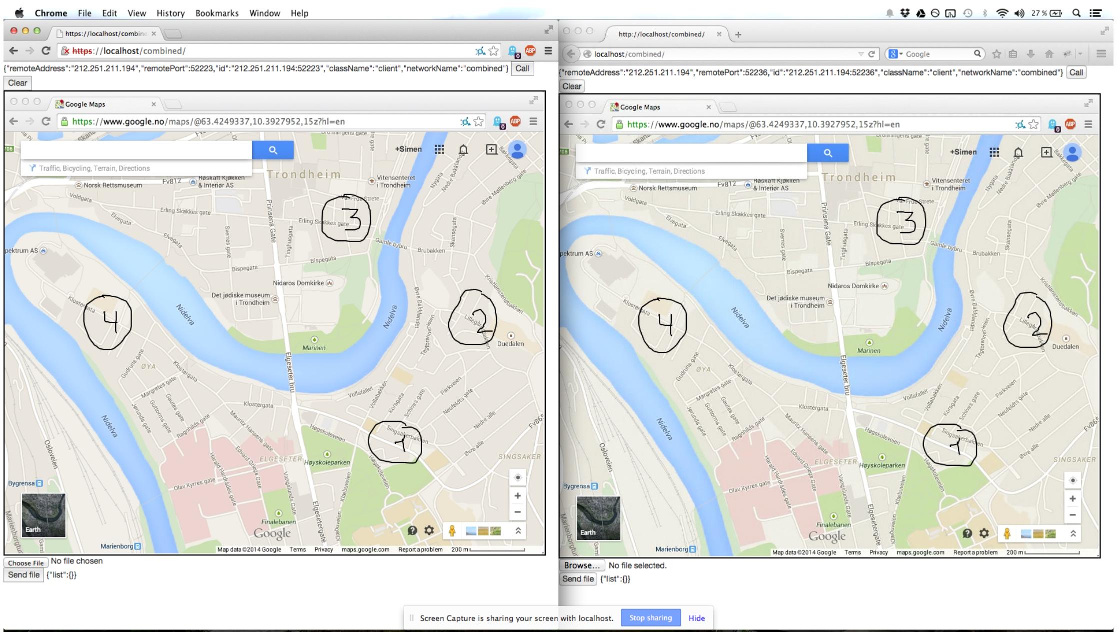

```
{"remoteAddress":"212.251.211.194","remotePort":52223,"id":"212.251.211.194:52223","className":"client","networkName":"combined"}
```
```
{"remoteAddress":"212.251.211.194","remotePort":52236,"id":"212.251.211.194:52236","className":"client","networkName":"combined"}
```
```
{"list":{}}
```
Screen Capture is sharing your screen with localhost.

FIGURE 5.10: Example of whiteboard and screen sharing combined using Google Chrome on the left and Mozilla Firefox on the right. Chrome has a separate window with maps.google.com open, and shares this window with Firefox. Both parties can draw on top of the map

5.2.4 Creating an AngularJS module

To enable easy implementation of the whiteboard tool, I created a separate AngularJS module that can easily be included in a project.

```
var app = angular.module('app', ['whiteboard']);

app.run(function(whiteboard) {

        var channel; //connect to another peer

        //provide whiteboard with a function to send data
        whiteboard.setSender(function (message) {
                channel.send(message);
        });

        //deliver received data to whiteboard
        channel.onmessage = function(message) {
                whiteboard.onmessage(message);
        };

});
```

Listing 17: Implementing the whiteboard module

This is enough to set up the module correctly. To use it in a web page, there is an AngularJS directive that can be utilized.

```
<div style="position:relative">
        <video ss-whiteboard></video>
</div>
```

Listing 18: Using the whiteboard directive

This shows how to add the whiteboard on a video element. The whiteboard will then resize itself to the size of the video, and it is possible to draw on top of it. Any HTML element can be used as a basis for the whiteboard. Using the directive on an empty `<div>` element will create a basic whiteboard with nothing underneath.

5.3 File sharing

The sharing of files is mentioned in several of the use cases from chapter 4. In use case 4.1, online education, the professor uses file sharing to distribute relevant files e.g. slides or documents. In use case 4.3, realtor-customer, the realtor sends files that contain pictures and other interesting information about the houses the customer is interested in.

The prototype already has a file sharing feature. The prototype uses socket connections to the NodeJS server and sends the file to the server, and the server sends it to the user receiving the file. This is certainly a viable option, but it is not peer-to-peer. This means that the server has to do extra work, and the transmission times will be longer since all data must go through the server. Therefore I will create a peer-to-peer file sharing tool that can be used over WebRTC.

This section will start off with a simple example to show how file sharing can be used in WebRTC, and then step by step address issues with the design and improve upon them.

5.3.1 How to read, send and receive file

It is possible to use `FileReader` to read a file, available in both Chrome and Firefox. The WebRTC standard specifies that it is possible to send binary data over data channels[6]. However this is not supported in Chrome or Firefox yet. Therefore the binary data must be converted to a text based format. We can use `btoa`[26] to convert binary data to base64[26], which is a text based format. The following snippet illustrates how to load a file, and send it over a data channel.

```
var reader = new FileReader();

var file; //obtain file somehow
var channel; //connect channel to other peer

reader.onload = function(event) {
        var buffer = event.target.result;

        channel.send(btoa(buffer));
};

reader.readAsBinaryString(file);
```

Listing 19: Reading file and sending it

To receive the file you need to convert it back to binary data. Listing 20 can be used to convert from base64 to binary arrays, which can then be used to reconstruct the file.

```
function base64toByteArrays(b64Data) {
        var sliceSize = 512;

        var byteCharacters = atob(b64Data);
        var byteArrays = [];

        for (var offset = 0; offset < byteCharacters.length; offset += sliceSize) {
                var slice = byteCharacters.slice(offset, offset + sliceSize);

                var byteNumbers = new Array(slice.length);
                for (var i = 0; i < slice.length; i++) {
                        byteNumbers[i] = slice.charCodeAt(i);
                }

                var byteArray = new Uint8Array(byteNumbers);

                byteArrays.push(byteArray);
        }
        return byteArrays;
}
```

Listing 20: [27] Converting base64 back to binary data

To download the file from memory to the hard disk it is possible to use an anchor element. Just create a link with the file as the reference and click it using JavaScript;

```
var b64file; //obtain entire file from other peer
var byteArrays = base64toByteArrays(b64file);

var blob = new Blob(byteArrays, {type: "application/octet-stream"});

var link = document.createElement('a');
link.href = window.URL.createObjectURL(blob);
link.download = filename;
document.body.appendChild(link);
link.click();
```

Listing 21: Using an anchor element to download the file to disk

This is all it takes to send a file over data channels. However it is quite limited. The max file size is also the max size a data channel can send in a single packet. If the file is larger, the file will become corrupted in transit.

5.3.2 Splitting the file in several packets

The max size of a data channel packet is not defined in the WebRTC standard[6], but from my own experience an upper limit of 10 kilobytes works well in both Google Chrome and Mozilla Firefox. The following code modifies the `onload` function of the `FileReader`.

```
reader.onload = function(event) {
        var buffer = event.target.result;

        var array = btoa(buffer).match(/.{1,10000}/g); //split file into pieces

                for (var i in array) {
                        channel.send(array[i]);
                }
        };
};
```

Listing 22: Splitting the file and sending it in pieces

This code will split up the file and then send it piece by piece. Now we can send files larger than 10 kB. There are still some issues with the design, as the entire file is loaded to memory. If the size of the file being sent is larger than the memory of one of the computers participating, the browser will crash. This is not a big problem since the memory of modern computers is usually several gigabytes, but sending large files can be necessary some times. Therefore I will continue to improve the design.

5.3.3 Improving memory performance sender side

If the sender tries to send the entire file at once, and the file size exceeds the memory of the computer, the browser will become unresponsive, and eventually crash. To fix this the file must be broken up into smaller parts, and read one by one. Listing 23 uses `file.slice` to get a Binary Large Object (blob)[28] that can be read and sent individually from the rest of the file.

The file can then be recreated at the receiving side in the same way as in section 5.3.1;

```
var maxSize = 100*1024*1024; //100 MB slize size

function readAndSend(blob, callback) {
        //read and send the blob as shown in the first example

        //wait until entire blob is sent
        if (callback) {
                callback()
        }
};

function senderLoop(i) {
        return function () {

                var blob;
                if (i + maxSize > size) {
                        blob = file.slice(i, size);
                        read(blob);
                }
                else {
                        blob = file.slice(i, i + maxSize);
                        read(blob, senderLoop(i+maxSize));
                }
        };
}
```

Listing 23: Splitting a file into several blobs to save memory

5.3.4 Improving memory performance receiver side

To prevent memory overflow on the receiving end, the file must be saved to the disk piece by piece. When the entire file is received the file can be reconstructed. This is possible using the File System API[29]. This allows the browser to access a sandboxed part of the disk. When the entire file has been saved in this sandbox, the file can be downloaded similar to section 5.3.1. This API is only available in Chrome, and not Firefox[30].

Listing 24 shows how to open the file system, and append a blob at the end of a file. When the writing is finished, the file can be downloaded from the sandbox just like shown in listing 21. It is worth noting that the file system in the example is temporary. This means that the browser can at any time delete your data. This can happen if the file you are trying to save is too large. A workaround for this is to use `window.PERSISTENT` when requesting the file system. The user of the application will then have to manually accept the use of the file system.

```
var blob; //The data to be written to file
var fileSize;
var fileName;

window.webkitRequestFileSystem(window.TEMPORARY, fileSize, successCallback,
        function(error) {});

function successCallback(fileSystem) {
        fileSystem.root.getFile(fileName, {create: false}, function(fileEntry) {
                fileEntry.createWriter(function(fileWriter) {

                        fileWriter.onwriteend = function(e) {
                                //download file
                        };
                        fileWriter.seek(fileWriter.length); //go to end of file
                        fileWriter.write(blob); //append the blob
                });
        });
}
```

Listing 24: Using File System API to save memory

5.3.5 Creating an AngularJS module

I implemented the file sharing tool using the basic principles described in previous sections. To make it as modular and independent as possible I created a separate AngularJS module that contains the file transfer tool. It is very easy to implement in another project. Listing 25 shows how to include `fileTransfer` in your angular module, and how to set up fileTransfer.

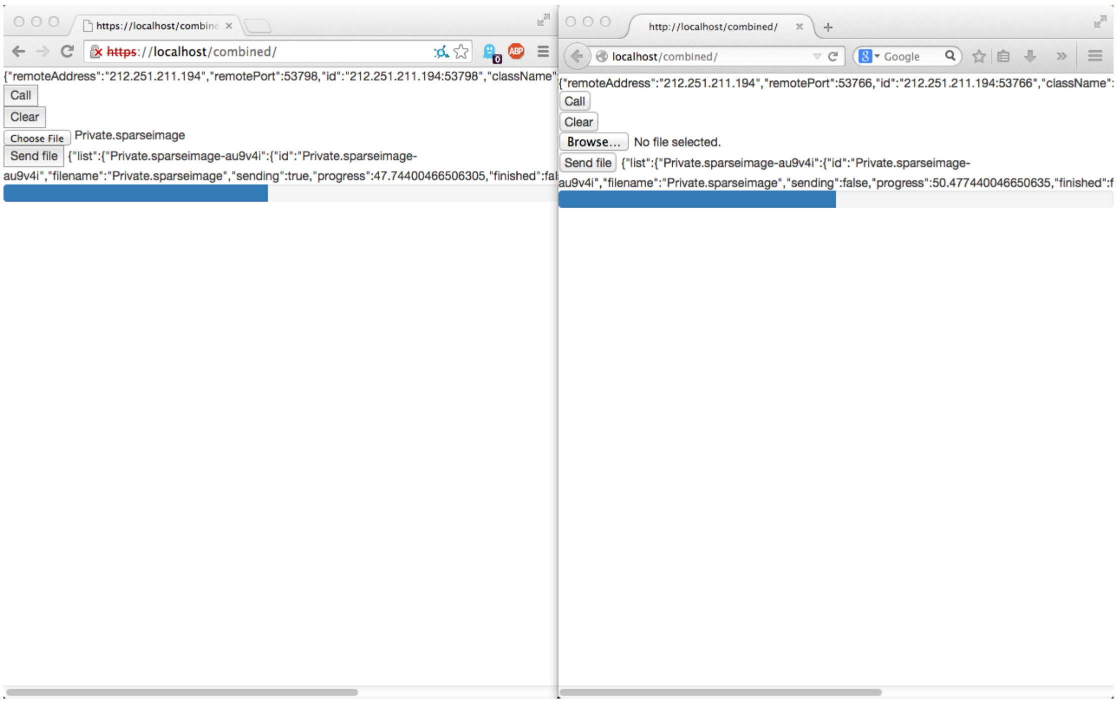
This module works with both Firefox and Chrome. The max file size that can be sent is up to the receiver, assuming the file is already stored on the disk of the sender. When the receiver is using Chrome, the max size is half of the free disk space available, since the file needs to be stored twice. Once in the sandbox, and once on the main disk. If the receiver uses Firefox, the maximum file size depends on the available free memory.

When using this tool you will provide it with a way to send and receive messages. This tool is designed with WebRTC in mind, but it is possible to use this file sharing tool over any kind of communication channel, not only WebRTC.

The progress of each file is stored in the `fileList` factory. This is a factory that contains real-time information about all the file transfers.

```
var app = angular.module('app', ['fileTransfer']);

app.run(function(fileTransfer) {

        var channel; //connect to another peer
        var file; //obtain file first

        var transfer = fileTransfer.newTransfer();

        //provide transfer with a function to send data
        transfer.setSender(function (message, callback) {
                channel.send(message);

                //use callback('sent') after the message was successfully sent
                //use callback('failed') if the message was not successfully sent
                if (callback) {
                        callback('sent');
                }
        });

        //deliver received data to transfer
        channel.onmessage = function(message) {
                transfer.onmessage(message);
        };

        transfer.sendFile(file);
});
```

LISTING 25: How to use the file transfer tool



```
{"remoteAddress":"212.251.211.194","remotePort":53798,"id":"212.251.211.194:53798","className":
Call
Clear
Choose File  Private.sparseimage
Send file  {"list":{"Private.sparseimage-au9v4i":{"id":"Private.sparseimage-
au9v4i","filename":"Private.sparseimage","sending":true,"progress":47.74400466506305,"finished":fal
```

```
{"remoteAddress":"212.251.211.194","remotePort":53766,"id":"212.251.211.194:53766","className":
Call
Clear
Browse...  No file selected.
Send file  {"list":{"Private.sparseimage-au9v4i":{"id":"Private.sparseimage-
au9v4i","filename":"Private.sparseimage","sending":false,"progress":50.477440046650635,"finished":f
```

FIGURE 5.11: Figure showing an ongoing file transfer from Chrome to Firefox

6. Discussion and conclusion

This chapter aims to wrap up the thesis. Section 6.1 presents some of the problems that complicated the work on the thesis. A discussion part is found in section 6.2. The future work in section 6.3 proposes ways to continue the research. Finally there is the conclusion in section 6.4.

6.1 Issues

This section will describe some of the issues that appeared during the work on this thesis, because of the rapidly changing technologies involved.

6.1.1 Chrome update M37 and data channels

When Chrome updated to version M37 the way data channels was handled was changed. From the release notes for Google Chrome update M37[31] from September 18 2014: "Issue 2645:- Fixed bug in SCTP data Channel. Now we close the datachannel when the send buffer is full or there are transport errors."

This change is concerning how the channel shall react when there is a buffer overflow. Before update M37 the buffer would throw an exception when the buffer was full, but the channel would remain open. Listing 26 shows how it was possible to send large amounts of information.

However the WebRTC standard specifies: "Attempt to send data on channel's underlying data transport; if the data cannot be sent, e.g. because it would need to be buffered but the buffer is full, the user agent must abruptly close channel's underlying data transport with an error"[6].

After update M37, Chrome follows the standard in this case. Listing 27 shows how to avoid buffer overflow post M37.

The timeout in both cases is set to 100 milliseconds. This was arbitrary decided for this example. If the timeout is lower it could result in faster transmission times, since

```
var queue = []; //fill queue with data to be sent
var dataChannel; //obtain a working data channel

function senderLoop() {
        while (queue.length) {
                try {
                        dataChannel.send(queue.shift());
                }
                catch (error) {
                        setTimeout(senderLoop, 100);
                        return;
                }
        }
}
```

Listing 26: Avoiding overflow on data channels pre M37 using error handling

```
function senderLoop() {
        while (queue.length) {
                if (dataChannel.bufferedAmount === 0) {
                        dataChannel.send(queue.shift());
                }
                else {
                        setTimeout(senderLoop, 100);
                        return;
                }
        }
}
```

Listing 27: Avoiding overflow on data channels post M37 using `bufferedAmount`

less time would be spent waiting. However this would imply that more time is spent checking to see if the channel is free and more timeouts, as a lower timeout would mean faster overflow. The sweet spot depends on the hardware of the host machine. Lower timeout implies faster transfer times, but could make the system more unresponsive.

6.1.2 Chrome update M37 and Dialogic XMS 2.2

Update M37 caused another problem as well. It affected the way Dialogic XMS 2.1 needed to communicate with Google Chrome. Dialogic released a new version short after. From the release notes of Dialogic XMS 2.2: "The ice-pwd field of the SDP produced by XMS is too short for the length of characters"[32].

However the prototype is still not working after the M37 chrome release. And Dialogic has not fixed the problem in subsequent updates. The team at Gintel AS has currently not been able to solve the problem.

6.2 Discussion

The WebRTC technology is still being implemented and it is changing rapidly, as shown in section 6.1, therefore it is volatile and can be difficult to include in your own stable software. In other branches of software development you can often choose which version of a program you want to work with, ignoring future updates for a current working version. When it comes to development for the web, security plays a big part and you usually want the latest version of the browser. Google Chrome will update automatically whenever an update is available[33]. Therefore you cannot pick and choose which version you want to use, and are constrained to the latest version available. This can create a period of time where your software is unusable, in the time from a new update to the point where you manage to adapt your program to fit the new update. This could result in unscheduled downtime of several days. Whether this is acceptable or not is a decision left for the individual projects.

Section 6.1.2 describes a problem that came with an update for Google Chrome. This problem was not solved at the time of writing, and because of this, the prototype was not ready to be extended by me during the time I worked on my thesis. Therefore I sought to make the tools I designed as easy to implement as possible, since I would not be the person to implement the features in the actual prototype. To achieve this I tried to make the components independent and self sustained, only requiring basic configuration to work.

A great benefit of using WebRTC in a web browser compared to Skype or a similar software is that a web browser is an essential part of the way modern computers are used. It is very likely that someone already has a browser installed on their computer, whereas you would have to download and install Skype. This makes an application using a web browser easier to access, and quicker to launch. If the application is used often on a computer, it might make more sense to have an external program installed on the computer as opposed to navigating to a specific web page every time. There are pros and cons, but with more options available to the developer, there are more ways a program can be crafted to suit specific needs.

Any implementation of WebRTC is required to use encryption when sending data[6]. This is beneficial for the security of the users. All data between two users is sent directly peer-to-peer, making any sort of Man-in-the-middle[34] attack difficult. When data is sent through the Internet there is no guarantee what path it will take, and to intercept it you need access to one of the routers on the path. Even if you were able to intercept the communication, the data would still be protected by encryption. WebRTC is native to the browser, therefore it is not necessary to trust closed-source third-party plugins with sensitive information. Because of these elements WebRTC is a welcome addition to the web security ecosystem.

6.3 Future work

This section will list some of the work that can be addressed in the future, to continue the work that was done in this thesis.

6.3.1 Implementing the features

Because of the problem described in section 6.1.2 the features designed and created in this thesis were not implemented into the prototype system. The tools are designed to be very modular and easy to implement in any project. Still the actual integration of the tools with the prototype remains.

6.3.2 Improving the whiteboard tool

To address the concurrency issues with the whiteboard tool described in section 5.2.2.1, an algorithm called Operational Transformation[35] could be helpful. It is a technology for enhancing collaboration applications. The basic concept is to convert conflicting operations using different techniques. Further research into this algorithm could decide whether or not this is applicable or too complex for this application.

6.3.3 Multiple peer connections for file sharing

When I experimented with file sharing I discovered that it was possible to transfer data faster between peers, if using several peer connections, and alternating between them when sending data, as opposed to using just one. This was experienced in Google Chrome. Is this behavior consistent with the WebRTC specification? Is it specific to Google Chrome or is it consistent across browsers? Is it a bug or is it intentional? This could be interesting to investigate further.

6.3.4 Browser compatibility

This thesis has focused mainly on Google Chrome and Mozilla Firefox. According to figure 2.1, Opera also has some support for WebRTC. Also, another browser has showed up, namely Bowser[36]. It is a WebRTC browser for iOS, created by Ericsson Research. Further research into the capabilities of these browsers could be interesting.

6.4 Conclusion

Unified communications aim to seamlessly integrate several communication platforms, where you only have to care about one platform instead of many. Combining different communication technologies can help productivity and create a more user friendly application. The prototype has accomplished unified communications by allowing both users of WebRTC and the telephony network to communicate together. Therefore it is possible to participate in video and audio conferences using either a web browser or a cell phone.

WebRTC can be a great addition to unified communications by adding new communication features in web browsers. WebRTC does not provide new functionality that was previously unavailable, but it includes native functionality that was previously only available through third party tools like Adobe Flash. Therefore WebRTC does not provide additional functionality to unified communication, but it improves upon the usability of the existing communication possibilities. The end user could end up with a smoother experience of the application if they do not need to handle any plugins.

WebRTC communication is required to be encrypted, and all data goes directly to the destination. Since WebRTC offers this functionality natively it is unnecessary to use third-party plugins. This is great for web security, and helps to make secure web solutions part of the norm and not the exception.

WebRTC offers reduced transmission times, reduced server load and not needing plugins. These are all great motivations to use WebRTC. Even though WebRTC is standardized, the browser implementations in Google Chrome and Mozilla Firefox are constantly changing. This makes developing with WebRTC very challenging since a working prototype can break when a new update is introduced. It is especially difficult when using third party tools for example Dialogic XMS, that also needs to be continuously updated whenever the browsers are updated. WebRTC is still very interesting and proposes many useful changes to the available tools when programming for web browsers. It is reasonable to assume that WebRTC will one day be completely implemented, and then it will be very useful. At this stage it is still problematic to use WebRTC for more than experimental demos.

In this thesis several features have been developed to further improve the existing prototype. These features focus on peer-to-peer communications rather than a centralized solution. A centralized system is needed for group conversations and multiplexing. However peer-to-peer is more beneficial for file sharing and real-time communication. Unforeseen problems stunted the work on this thesis and ensured the improvements were not integrated in the prototype. By using modular components in the AngularJS framework, the tools were created with ease of installation in mind.

A. Whiteboard prototype

Listing 28 illustrates how to make a functioning whiteboard. The basic concept is to use mouse listeners on a canvas element, and draw lines and dots according to the mouse movements. The information that is used to draw on the canvas is also sent over a data channel to another peer in real time. The code does not illustrate how to handle the received status update. This can be accomplished with calling `drawDot` or `drawLine` depending on the incoming message.

```
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext("2d");
var dataChannel; //connect to other peer and establish a data channel
var mouseDownFlag = false, prevX = 0, currX = 0, prevY = 0, currY = 0;

function drawLine(prevX, prevY, currX, currY) {
        ctx.beginPath();
        ctx.moveTo(prevX, prevY);
        ctx.lineTo(currX, currY);
        ctx.lineWidth = 2;
        ctx.stroke();
        ctx.closePath();
}
function drawDot(currX, currY) {
        ctx.beginPath();
        ctx.fillRect(currX, currY, 2, 2);
        ctx.closePath();
}
function updateDrawing(isDot) {
        prevX = currX;
        prevY = currY;
        currX = event.clientX - canvas.offsetLeft;
        currY = event.clientY - canvas.offsetTop;

        if (isDot) {
                drawDot(currX, currY);
        }
        else {
                drawLine(prevX, prevY, currX, currY);
        }

        var statusUpdate = {
                prevX: prevX, prevY: prevY,
                currX: currX, currY: currY, isDot: isDot,
        };
        dataChannel.send(JSON.stringify(statusUpdate));
}

canvas.addEventListener("mouseout", mouseListener, false);
canvas.addEventListener("mouseup", mouseListener, false);
canvas.addEventListener("mousedown", mouseListener, false);
canvas.addEventListener("mousemove", mouseListener, false);

function mouseListener(event) {
        var type = event.type;
        if (type == 'mousedown') {
                mouseDownFlag = true;
                updateDrawing(true);
        }
        else if (type == 'mouseup' || type == "mouseout") {
                mouseDownFlag = false;
        }
        else if (type == 'mousemove') {
                if (mouseDownFlag) {
                        updateDrawing();
                }
        }
}
```

Listing 28: Whiteboard prototype

B. Source code

Attached to the thesis are the source code for the tools created in chapter 5.

The screen sharing extension for Google Chrome described in section 5.1 is found in the folder `screenCapture-chrome-extension`.

The whiteboard AngularJS module described in section 5.2 is found in the folder `whiteboard-angular-module`.

The file sharing AngularJS modlue described in section 5.3 is found in the folder `fileTransfer-angular-module`.

Bibliography

[1] Skype. URL http://www.skype.com/en/.

[2] Adobe. Adobe flash runtimes. URL https://www.adobe.com/products/flashruntimes.html.

[3] Q-Success. Usage of flash for websites. URL http://w3techs.com/technologies/details/cp-flash/all/all.

[4] Xiao Chen. Unified communication and webrtc. Master's thesis, Norwegian University of Science and Technology, 2014.

[5] Ian Sommerville. *Software Engineering*, volume 9. Pearson, 2011.

[6] Adam Bergkvist, Daniel C Burnett, Cullen Jennings, and Anant Narayanan. Webrtc 1.0: Real-time communication between browsers. *Working draft, W3C*, 91, 2012.

[7] W3Counter. October 2014 market share. URL http://www.w3counter.com/globalstats.php?year=2014&month=10.

[8] Daniel C. Burnett, Adam Bergkvist, Cullen Jennings, and Anant Narayanan. Media capture and streams, 2013. URL http://www.w3.org/TR/mediacapture-streams/.

[9] M. Handley, UCL, V. Jacobson, and Packet Design C. Perkins University of Glasgow. Sdp: Session description protocol, July 2006. URL https://tools.ietf.org/html/rfc4566.

[10] Ilya Grigorik. High performance browser networking. URL http://chimera.labs.oreilly.com/books/1230000000545/ch18.html#_delivering_media_with_srtp_and_srtcp.

[11] M. Baugher, D. McGrew, Cisco Systems Inc., M. Naslund, E. Carrara, K. Norrman, and Ericsson Research. The secure real-time transport protocol (srtp). URL http://www.ietf.org/rfc/rfc3711.txt.

[12] GoogleChrome. Adapter.js. URL `https://github.com/GoogleChrome/webrtc/blob/master/samples/web/js/adapter.js`.

[13] Dialogic. Powermedia xms, . URL `http://www.dialogic.com/Products/media-server-software/xms.aspx`.

[14] Joyent Inc. Nodejs. URL `http://nodejs.org/`.

[15] Google. Html enhanced for web apps!, . URL `https://angularjs.org/`.

[16] Single-page application. URL `https://en.wikipedia.org/wiki/Single-page_application`.

[17] Google. What is a module?, . URL `https://docs.angularjs.org/guide/module`.

[18] Google. Providers, . URL `https://docs.angularjs.org/guide/providers`.

[19] Google. What is a directive?, . URL `https://docs.angularjs.org/guide/directive`.

[20] Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Carlisle Adams. X. 509 internet public key infrastructure online certificate status protocol-ocsp. Technical report, RFC 2560, 1999.

[21] 2014. URL `https://code.google.com/p/chromium/issues/detail?id=347641`.

[22] chrome.desktopcapture, . URL `https://developer.chrome.com/extensions/desktopCapture#method-chooseDesktopMedia`.

[23] Event pages api, . URL `https://developer.chrome.com/extensions/event_pages`.

[24] Google. Content scripts api, . URL `https://developer.chrome.com/extensions/content_scripts`.

[25] Chrome developer - message passing, . URL `https://developer.chrome.com/extensions/messaging`.

[26] W3C. Base64 utility methods, November 2014. URL `http://www.w3.org/html/wg/drafts/html/master/webappapis.html#atob`.

[27] Jeremy Banks. Creating a blob from a base64 string in javascript. URL https://stackoverflow.com/questions/16245767/creating-a-blob-from-a-base64-string-in-javascript.

[28] Jim Starkey and Ann Harrison. The true story of blobs., October 2000. URL http://web.archive.org/web/20110723065224/http://www.cvalde.net/misc/blob_true_history.htm.

[29] File api: Directories and system, 2009. URL http://dev.w3.org/2009/dap/file-system/pub/FileSystem/.

[30] File system api guide. URL https://developer.mozilla.org/en-US/docs/WebGuide/API/File_System.

[31] Chrome webrtc m37 release notes, . URL https://groups.google.com/forum/#!topic/discuss-webrtc/Qt99-FXzKkU.

[32] Dialogic. Dialogic® powermedia™ xms release 2.2, . URL https://www.dialogic.com/webhelp/XMS/2.2/XMS_ReleaseNotes.pdf.

[33] Google. Update google chrome, . URL https://support.google.com/chrome/answer/95414?hl=en.

[34] Tanmay Patange. How to defend yourself against mitm or man-in-the-middle attack. URL http://hackerspace.lifehacker.com/how-to-defend-yourself-against-mitm-or-man-in-the-middl-1461796382.

[35] Chengzheng Sun. Operational transformation frequently asked questions and answers. URL http://www3.ntu.edu.sg/home/czsun/projects/otfaq/#_Toc321146201.

[36] EricssonResearch. A webrtc browser for ios developed in the open. URL https://github.com/EricssonResearch/Bowser.