FH JOANNEUM (University of Applied Sciences)

WebRTC

Development of a browser based real-time peer-to-peer remote support application

Bachelor Thesis

submitted in conformity with the requirements
for the degree of
Bachelor of Science in Engineering (BSc)

Bachelor’s degree program **Internettechnik**
FH JOANNEUM (University of Applied Sciences), Kapfenberg

supervisor: Dipl. Ing. Johannes Feiner

submitted by: Michael Stifter
personal identifier: 1310418054

01 / 2016

Obligatory signed declaration:

I hereby declare that the present Bachelor’s thesis was composed by myself and that the work contained herein is my own. I also confirm that I have only used the specified resources. All formulations and concepts taken verbatim or in substance from printed or unprinted material or from the Internet have been cited according to the rules of good scientific practice and indicated by footnotes or other exact references to the original source.

The present thesis has not been submitted to another university for the award of an academic degree in this form. This thesis has been submitted in printed and electronic form. I hereby confirm that the content of the digital version is the same as in the printed version.

I understand that the provision of incorrect information may have legal consequences.

[illegible]

Michael Stifter Graz, 31.01.2016

Table of contents

Abstract ... 6
Kurzfassung ... 7
1 Introduction ... 8
2 Analysis of remote support applications ... 9
2.1 Criteria ... 9
2.2 Essential functional requirements ... 10
2.2.1 Audio and video stream ... 10
2.2.2 Overlay indicators ... 10
2.2.3 Gestures ... 10
2.2.4 Pause video feed ... 11
2.3 Connection architecture ... 11
2.3.1 Client-server ... 11
2.3.2 Peer-to-peer ... 12
2.4 Vision enhancing technologies ... 13
2.4.1 Augmented Reality ... 13
2.4.2 Mixed reality ... 14
2.5 Hardware ... 14
2.5.1 Handheld devices ... 14
2.5.2 Wearable devices ... 15
2.6 Software ... 16
2.6.1 Desktop applications ... 16
2.6.2 Mobile applications ... 17
2.6.3 Web-based applications ... 17
2.7 Result matrix ... 18
2.8 Research conclusions ... 19
2.9 Personal opinion ... 20
3 WebRTC ... 22
3.1 Overview ... 22
3.1.1 Architecture ... 22

3.1.2 Functionality and features 23
3.1.3 History 24
3.1.4 Advantages 24
3.1.5 Limitations 26
3.1.6 Current status 27
3.2 API components 27
3.2.1 MediaStream 27
3.2.2 PeerConnection 28
3.2.3 DataChannel 28
3.3 Connection setup 29
3.3.1 Signaling 29
3.3.2 NAT problem 29
3.3.3 ICE candidates 30
3.3.4 Session description offers and answers 31
3.3.5 Data channels 32
4 Prototype 35
4.1 Management server 35
4.1.1 Implementation 35
4.1.2 Web server 35
4.1.3 WebSockets 35
4.1.4 Management and control tasks 36
4.2 Web interface 36
4.2.1 HTML5 elements 36
4.2.2 User interaction 36
4.2.3 Responsive design 36
4.2.4 Facilitating libraries 37
4.3 Overlay indicators feature 37
4.3.1 Implementation 37
4.3.2 Mouse events 38
4.3.3 Touch events 39
4.3.4 Data transfer 40
4.3.5 Text chat 40

5 Evaluation 41
5.1 Setup 41
5.2 Method 41
5.2.1 Media constraints 41
5.2.2 Google Chrome Developer Tools 42
5.2.3 User tests 43
5.3 Results 43
6 Outlook 45
6.1 Screenshots 45
6.2 User authentication 45
6.3 E-mail invitations 45
6.4 Cross-platform application 46
6.5 Sessions with more than two users 46
7 Conclusion 47
List of tables 49
List of figures 50
List of listings 51
List of abbreviations 52
Bibliography 53

Abstract

Web Real-Time Communication (WebRTC) enables internet users to communicate with each other directly over encrypted audio and video streams, without the need of servers transferring the data traffic. This peer-to-peer technology can be used in web browsers, without installing additional software or plugins. This thesis proposes a browser-based remote support application that enables users to communicate with each other over audio and video streams and assist one another via drawing overlay indicators on top of the video feed. Remote support applications offer a fast and convenient method to repair malfunctions in factories, without demanding physical presence of experts on site. In the first part of the thesis are important terms and criteria regarding remote support applications defined and several technological options for implementing such an application compared. Second, WebRTC is presented as a possible technology in more detail and its advantages and limitations discussed. Based on the outcome of the research, a prototype application was developed and the key findings of the development process are explained. The finished application was evaluated with regard to the minimum level of video resolution and network quality necessary to be perceived as helpful by users. The evaluation process showed that a video resolution of 640 x 360 pixels and a network throughput of at least 750 kbit/s is necessary for the application to be perceived as helpful by the users. The end of the thesis addresses possible further extensions of the prototype.

Kurzfassung

Web Real-Time Communication (WebRTC) bietet Internet-Nutzern die Möglichkeit, miteinander über verschlüsselte Audio- und Video-Streams zu kommunizieren. Dafür sind keine Webserver notwendig, der Datentransfer geschieht direkt, oder peer-to-peer. Das Besondere an dieser Technologie ist die Tatsache, dass sie direkt in einem Webbrowser verwendet werden kann, ohne zusätzliche Software installieren zu müssen. In dieser Bachelorarbeit wird eine browser-basierte Fernunterstützungsanwendung vorgestellt, die es den Benutzern ermöglicht, miteinander über Audio- und Video-Streams zu kommunizieren und sich gegenseitig mithilfe von Zeichnungen, die über dem Videobild dargestellt werden, unterstützen zu können. Fernunterstützungsanwendungen können in Fabriken eingesetzt werden, um defekte Maschinen in einer einfachen und schnellen Weise zu reparieren, ohne dabei die Präsenz eines Experten vor Ort zu erfordern. Zu Beginn der Arbeit werden wichtige Begriffe und Kriterien betreffend Fernuntersützungsanwendungen erläutert und mehrere technologische Ansätze für die Entwicklung einer solchen Anwendung verglichen. Danach wird WebRTC als eine mögliche Technologie näher vorgestellt und die Vor- und Nachteile aufgezeigt. Auf Basis der vorgenommenen Recherche wurde eine prototypische Anwendung entwickelt und die aufschlussreichsten Erkenntnisse aus diesem Prozess erklärt. Danach wurde der Prototyp hinsichtlich der minimal notwendigen Netwerk- und Videoqualität untersucht, um für die Benutzer als unterstützend und hilfreich wahrgenommen zu werden. Die Benutzerbefragungen ergaben, dass eine Video-Auflösung von 640 x 360 Pixel und eine Netzwerkverbindung mit zumindest 750 kbit/s Datendurchsatz notwendig ist, um als hilfreich wahrgenommen zu werden. Abschließend werden mögliche zukünftige Erweiterungen für die Anwendung beschrieben.

1 Introduction

Assistive technology has been in use in factories for a few years now. Also known as remote support applications, they enable on-site personnel to repair malfunctions under support of experts, while they are connected via audio and video streams. For companies, this technology is quite advantageous as disruptions can be repaired significantly faster, without the necessity of an expert having to be physically present.

However, such proprietary video chat applications (i.e. Skype[1]) do have a major disadvantage: Data flows over a third-party server. Companies, particularly those dealing with sensitive data might not want that, as they can never be sure that their data does not fall into the wrong hands. In terms of performance, these applications further suffers from another drawback: higher network latency is required for the data transfer.

To eradicate these problems, Web Real Time Communication (WebRTC) could be used instead of a proprietary video chat application. With WebRTC, a server is only needed for users to find each other and set up a connection. After that, the data is going directly from user to user, or peer-to-peer, thus reducing network latency significantly. Furthermore, data encryption is mandatory for all components of WebRTC.

This thesis is structured as follows: In the first part, the key features of remote support applications are described, an overview of possible technologies to implement such applications is provided. In addition, similar research with regard to this field of study will be compared. In the second part, WebRTC is presented as a potential implementation method by highlighting its advantages and disadvantages. In a third step, the essential insights regarding the development of a prototype application are addressed. Chapter 5 discusses the results of the evaluation of the prototype application. The conclusion section summarises the findings and experiences of the prototype development and outlines possible ways of extending the current work.

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

2 Analysis of remote support applications

In factories, troubleshooting malfunctioning machines can be an arduous task. For handling complex devices and their performance issues, it is often necessary to conduct the repair or maintenance because local factory personnel is not trained to fix the problem. In the past, this required physical presence of the expert on site. This has changed with the era of digital technology. In most cases, nowadays, it is sufficient for the skilled employee to assist a technician, who usually does not possess the required know-how in this field, via video and audio streams. This form of remote troubleshooting has significantly reduced the costs for travel and work completion (cf. Chen et al. 2013).

This chapter will take a deeper look at necessary requirements of remote support applications, their essential features as well as the technology and hardware that are required to implement them. Finally, a conclusion will be drawn how a remote support application in a factory setting can be implemented to exploit its full potential.

2.1 Criteria

Huang et al. (2013) define several criteria that remote support applications need to fulfil in order to be beneficial: First, it is necessary for the cooperating parties to communicate with each other verbally. Second, the consulted expert should be able to have the same visual field as the worker to be able to provide a maximum level of assistance. Furthermore, „the helper should be able to point to the objects in the workspace and use […] hand gestures to guide the worker“ (Huang et al. 2013), which the worker should in turn be able to see. Ideally, both hands of the worker are free to enable the performance of physical tasks under a minimum of confinement in the working environment and, additionally, increase the safety conditions for the worker.

To achieve a satisfying level of portability, remote support applications should be independent of specific hardware platforms or operating systems. This enables users to move around the factory, provided they are equipped with a mobile device. Additionally, it is essential that users are not required to use only one specific type of device, but rather use the one that suits them best for the underlying task. Another important issue in this context is the support of multiple audio and video encodings, as different device types might have access to different media formats (cf. Reinhard et al. 1994, p. 30).

2.2 Essential functional requirements

According to Reinhard et al. (1994, p. 29), the interaction between the connected parties belongs to the most important functional requirements in a remote collaboration environment. They distinguish between implicit and explicit interaction. Communicating over text messages or images is classified as implicit interaction, while audio and video chat conferencing belongs to explicit interaction. In the following section, selected requirements with regard to interaction between collaborating users are described in more detail.

2.2.1 Audio and video stream

Huang et al. (2013) argue that a setting where the participating parties are able to hear and speak to each other is substantially more effective than communicating over text messages. They proposed the most efficient workspace setup to be one where the helper has a „panoramic view of the worker's workspace" (Huang et al. 2013). This is essential in maintaining overall awareness of the working environment. The worker, on the other hand, does not need to see the helper or his environment on screen, but instead his own video feed enhanced with overlay indicators suggesting possible solutions to a task. Additionally, they should able to communicate over a wireless network using microphones and speakers.

2.2.2 Overlay indicators

It is possible that some technical details are difficult to explain verbally, even when the helper has a complete understanding of the solution to the problem. Chen et al. (2013) describe this situation as an „uneven […] knowledge distribution" between the two involved parties. A solution to this problem could be the opportunity to draw simple sketches on the screen, which are transferred to the other person's screen and subsequently rendered on top the video feed. These overlay indicators can significantly improve the mutual understanding between the helper and the worker concerning the problem at hand and possible remedies (cf. Chen et al. 2013).

2.2.3 Gestures

Chen et al. (2013) propose another solution to improve the understanding in the assistance process: the display of hand gestures in addition to the previously described overlay indicators. This can be particularly useful when there is a large number of different components on the screen or the worker has a limited amount of knowledge about the malfunctioning system. In that case, hand gestures from the helper can be a valuable addition to verbal comments and overlay indicators. Possible gestures include pointing to specific objects, holding and pressing, wiping as well as turning or screwing objects in certain directions.

While this feature is indisputably helpful, its implementation, however, cannot be considered trivial. It requires capturing the helper’s hand on top of a black background with a separate camera. Afterwards, the image is processed with a grey-scale function and, subsequently, converted into a mask image, where pixels below a certain threshold value are set to black, and those above are set to white. Finally, all black pixels are becoming transparent, and the resulting image can be shown on top of the video feed of the worker (cf. Chen et al. 2013).

2.2.4 Pause video feed

An additional essential feature is the possibility to pause the video feed. A constantly running video stream would aggravate the assistance of the helper immensely, because the helper would have to draw the indicators on a moving, most likely unsteady image. This is unacceptable in terms of usefulness for both the helper and the worker. Consequently, a remote support application must feature the possibility to freeze the video feed in order to provide a maximum level of assistance (cf. Chen et al. 2013).

2.3 Connection architecture

Broadly speaking, possible connection architecture for remote support applications can be divided into two types: client-server and peer-to-peer. Both will be explained in more detail in this section.

2.3.1 Client-server

In a client-server network, both parties exchange data over a network connection. As depicted in Figure 1, multiple clients can request files from the server, for instance via protocols such as Hypertext Transfer Protcol (HTTP), or any other protocol the server can process. The server’s task is to fulfil the request of the clients. In other words, the server *responds* to it. In a standard HTTP environment, this principle is called request-response. (cf. Sinha 1992, p. 78f).

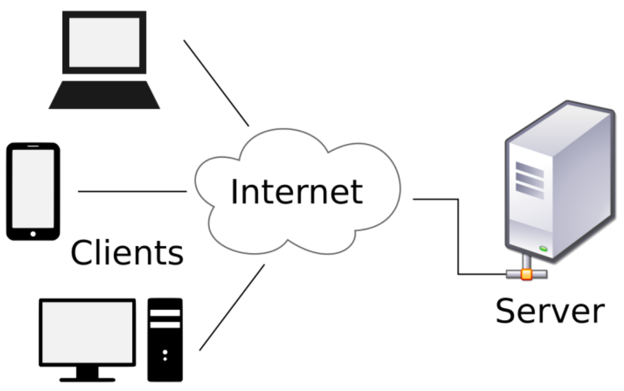


Figure 1: Multiple clients, which can be of different device types, communicate with a server over the internet[2]

While HTTP is a stateless protocol, it is possible to maintain a full-duplex connection between client and server since the introduction of WebSockets. The standard HTTP connection is upgraded to a WebSocket connection with an additional initial handshake, which enables WebSockets to use the same port as the HTTP connection. A significant advantage of this approach is the low latency for network transfer (cf. Davies, Zeiss & Gabner 2012).

For the implementation of a remote support application using this architecture, this implies that when two clients want to exchange data like audio and video streams, the server must implement the logic to facilitate this functionality. The main tasks are setting up a connection between two users and transfer the received data from one client to the other. It has to be noted, that in the scenario at hand there is no actual connection between the users but rather separate client-server request and response connections handled by the server. To the clients, however, it appears like they are connected together directly, while in reality the data is transferred over the server. Consequently, the latency for the network transfer is higher, even when WebSockets are used for maintaining connections between the server and the clients.

2.3.2 Peer-to-peer

In comparison, clients are linked together directly, over the shortest available network path when using peer-to-peer connections. This setup results in lower latency for the network transfer compared to a client-server architecture. It has to be noted, however, that in order to set up a peer-to-peer connection, a management

[2] https://upload.wikimedia.org/wikipedia/commons/thumb/c/c9/Client-server-model.svg/2000px-Client-server-model.svg.png [27 January 2016]

server is still necessary for the clients to find each other and start the connection. After the connection setup, the data is exchanged directly, without passing the server. This paradigm benefits remote support applications because they are inherently data-intense due to the constant amount of data transfer of audio and video streaming.

One peer-to-peer technology that has emerged over the past few years is WebRTC. It enables web browsers and mobile applications to share peer-to-peer connections without the installation of additional software or plugins. This offers a substantial advantage to web developers, who are now able to implement real-time communication applications such as video chats for web browsers using a JavaScript Application Programming Interface (API). Special knowledge about telecommunication technology is not required.

2.4 Vision enhancing technologies

As already indicated in chapter 2.2.2., overlay indicators can significantly improve the usefulness of a remote support application. To implement this functionality, a vision technology like Augmented Reality or mixed reality could be used. The characteristics of these two technologies are discussed in the following chapters.

2.4.1 Augmented Reality

Augmented Reality bridges the gap between the real world and Virtual Reality, a world that is entirely generated by computers. It does so by enhancing the senses of human beings, most commonly the visual sense, though less frequently also the sense of hearing or smell (cf. Bonsor n.d.). In other words, „Augmented Reality adds graphics, sounds, haptic feedback and smell to the natural world as it exists“ (Bonsor n.d.). Popular applications like Wikitude[3], for instance, recognise tourist attractions by analysing the user’s camera feed and label them with information about it on the screen. Most commonly, there is a tracking engine that examines still frames of the video feed and matches them against specific patterns. This technique originates from the field of computer vision.

Despite their numerous beneficial features, Augmented Reality applications suffer from several disadvantages. First, they commonly need some form of visual markers in order to be recognised by the tracking engine. The markers need to be placed on the object that should be augmented, which results in administrative effort. Some

[3] http://www.wikitude.com/

Augmented Reality frameworks, like Vuforia[4] for instance, offer object recognition simply by scanning its outline. However, this technology could still prove to be complicated to use due to the necessity of a high resolution camera that can handle insufficient or mixed lighting conditions, which could be expected in factories.

2.4.2 Mixed reality

Mixed reality offers a flexible combination of Augmented and Virtual Reality. In mixed reality, users can see computer-generated objects in addition to the real world. Admittedly, the line between Augmented Reality and mixed reality can be described as vague at best and might have only been created for marketing purposes (cf. Johnson 2015). The main difference between them is that in mixed reality the computer-generated objects are not triggered by analysis of image features, but rather by a different, external source, for instance by a human being drawing overlay indicators on the screen.

2.5 Hardware

There are several different types of hardware that can be utilised in the context of remote support applications. As stated above, an essential characteristic is the ability to move around freely. As a result, only mobile devices qualify to be used by the worker. In contrast, the helper could also use desktop computers, as there is no imminent need for him to move around. Broadly speaking, there are two main groups of mobile devices, which need to be discussed in detail, namely handheld and wearable devices.

2.5.1 Handheld devices

The term handheld devices refers to all electronic telecommunication devices that can be carried around and, as the name implies, held in a hand. Handheld devices are in most cases equipped with a display screen that is able to process touch inputs. Most commonly, handheld devices are smartphones and tablets. For remote support applications, both types can be used. Tablets have a slight advantage of larger screen sizes, while smartphones, in comparison, are less disruptive as they demand less space when carried around. All handheld devices, however, prevent workers from using both hands, thus limiting them in carrying out their work without distractions.

[4] https://www.qualcomm.com/products/vuforia

2.5.2 Wearable devices

Figure 2: Google Glass[5]

In brief, a wearable device can be described as electronic technology which can be worn on the body, without the need of holding it (cf. Tehrani & Andrew 2014). Over the last years, there was a vast amount of newly presented wearable devices, ranging from smart glasses such as Google Glass (see Figure 2) to smart watches, like the Apple Watch (see Figure 3). While watches do not meet the requirements for being a part of a remote support applications due to the lack of a camera and a significantly limited screen size, smart glasses, on the other hand, can be employed to be used in such a setting. They have a limited screen size compared to tablets but come with the substantial advantage that the worker can perform tasks using both hands.

Figure 3: Apple Watch[6]

Huang et al. (2013) propose an alternative to the previously described devices, as they are rather targeting the consumer market than specific industries and environments. Their solution consists of a common safety helmet that is equipped with a camera capturing the visual field of the worker, a near-eye display as well as

[5] http://www4.pcmag.com/media/images/423989-google-glass.jpg?thumb=y [27 January 2016]
[6] http://tr3.cbsistatic.com/hub/i/r/2014/09/15/0f4927f9-6ef6-467a-83ac-6ea6c205c028/resize/620x/cf656324db685ab0aa85e0ffbfd548d9/applewatch-hero.jpg [27 January 2016]

a microphone and headphones for speech communication with the helper. They decided to use a near-eye display because it enables users to keep a clear view of the surrounding environment, which is particularly important in safety-critical settings like production factories. (cf. Huang et al. 2013).

2.6 Software

In this section, three possible application types for implementing a remote support application will be discussed: Desktop applications, mobile applications and web-based applications.

2.6.1 Desktop applications

A desktop application is software that runs autonomously on a PC or laptop. It needs to be installed on a device before it can be executed. If one or more parts of the software need to be changed, a new version or update of the software has to be installed. Typical desktop applications include text processors and video or music players (cf. Smith n.d.). Popular programming languages for the development of desktop applications are Java and Microsoft .NET. While applications written in Java can be used on any desktop device that has Java installed, those written in Microsoft .NET can only be executed with their full potential on devices that run on the Microsoft Windows operating system.

One approach to develop a peer-to-peer desktop application is to use Java Sockets and Remote Procedure Calls (RPC). A Java web server such as WildFly[7] is required, where clients could connect to and open socket connections with the server. If two clients want to establish a peer-to-peer connection between them, the server has to pass on the respective peer’s socket connection. This approach, however, requires the implementation of programming logic that handles audio and video encodings, quality of service, screen resolutions and frame rate autonomously, which can be considered as rather complex. Holliday, Houston & Jones (2008) point out that this approach has received less appraisal since the introduction of web services. According to them this is due to the fact that on the one hand web browsers are ubiquitous nowadays and on the other hand HTTP packets do not commonly suffer from firewall limitations as RPC do.

[7] http://wildfly.org/

2.6.2 Mobile applications

Mobile applications are computer programs that run on mobile, mostly handheld devices like smartphones and tablets. Similar to desktop applications, they run natively within the device's operating system. Native mobile applications present developers with the problem that the applications have to be separately implemented for the different operating systems, for instance in Java for Android or in Swift for iOS.

There are several frameworks, such as Apache Cordova[8] for instance, which try to solve this problem by offering developers the option to generate applications for smartphones from a single code base to a variety of operating systems. Typically, these cross-platform applications are written in a similar way as web-based applications, using HTML, JavaScript and Cascading Style Sheets (CSS), which is then displayed in a special web-view layer within an operating system specific native application. While this offers developers more flexibility regarding the portability of their application and a significantly reduced time required for the implementation, cross-platform applications typically cannot exploit the operating system's full potential regarding access to the device's sensors or the operating system specific user interface, due to their one-size-fits-all approach (cf. Ciman, Gaggi & Gonzo 2014).

2.6.3 Web-based applications

In contrast to the previously described application types, web-based applications do not need to be locally installed on the device, but are rather accessed through a web browser. They consist of HTML pages, which can either be static files or dynamically created ones, with additional JavaScript and CSS files to extend and improve the web page's layout and functionality. Typically, web-based applications are built on the client-server principle. Clients request the web page from the server, and displays it in the browser upon response arrival.

Compared to desktop applications, web-based applications offer the advantage that they can be maintained with less effort, since the software only needs to be changed once on the server it runs on, instead of each device that has installed it. Furthermore, web-based applications can be used from anywhere as long as the device is connected to the internet, while desktop applications are physically

[8] https://cordova.apache.org/

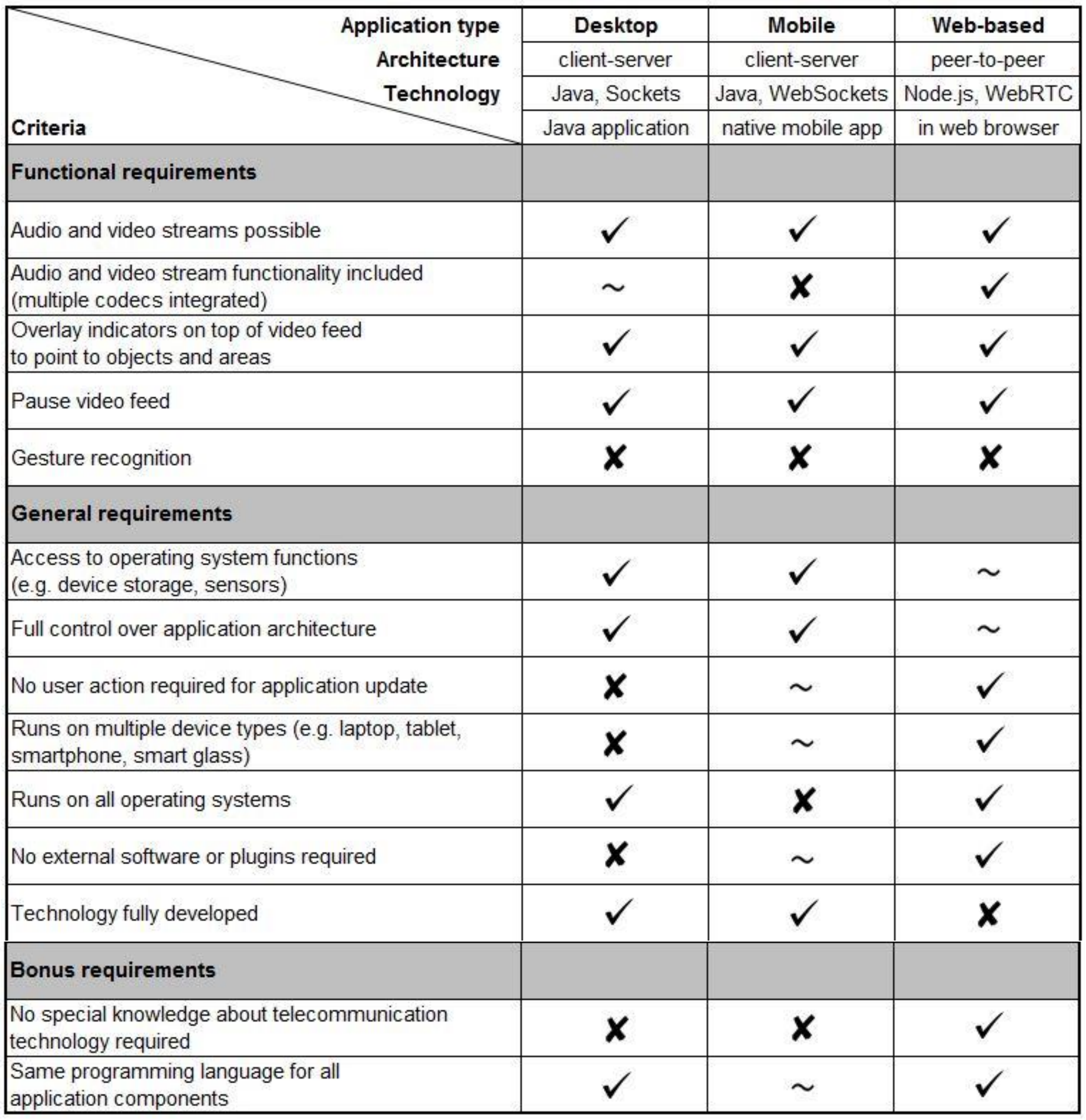
constrained to the device they are installed on. In contrast, however, desktop applications are less vulnerable to security risks, due to their confined environment (cf. Smith n.d.).

One approach that has received a substantial amount of attention over the last years is Responsive Web Design. Its aim is the „creation of web sites that take into account different types of devices, usually from mobile phones to desktops, and optimise viewing experience for the device at hand“ (Voutilainen & Salonen 2015). This is achieved by using flexible grids and images in addition to CSS3 media queries. With media queries, it is possible to create different CSS rules depending on specific attributes of the device, like screen size or the current orientation of the device (cf. Johansen, Pagani Britto & Cusin 2013). This enables developers to write web applications that can be used on devices with large screens as well as devices with very small screens, like smartphones or even wearables such as smart glasses, without considering the device’s operating system.

2.7 Result matrix

In order to find a suitable application type for a remote support application, all three previously described application types are compared with regard to the conducted research and the criteria discussed in Chapter 2.1. General requirements which are beneficial in terms of developing computer applications are also added to the comparison. These requirements include access to operating system specific functions like sensors and storage on the device or the possibility to automatically deploy application updates. Furthermore, the degree is assessed to which the technology is fully developed and if it is possible to use the application on multiple operating systems without additional effort.

Compared are a client-server desktop application using Java Sockets and a Java application on the cilent side, a mobile application based on a client-server architecture and WebSockets for the server part and a native app for the client part, as well as a web-based peer-to-peer application using WebRTC. The results of the comparison are displayed in Table 1.



Application type	Desktop	Mobile	Web-based
Architecture	client-server	client-server	peer-to-peer
Technology	Java, Sockets	Java, WebSockets	Node.js, WebRTC
Criteria	Java application	native mobile app	in web browser
Functional requirements			
Audio and video streams possible	✓	✓	✓
Audio and video stream functionality included (multiple codecs integrated)	~	✘	✓
Overlay indicators on top of video feed to point to objects and areas	✓	✓	✓
Pause video feed	✓	✓	✓
Gesture recognition	✘	✘	✘
General requirements			
Access to operating system functions (e.g. device storage, sensors)	✓	✓	~
Full control over application architecture	✓	✓	~
No user action required for application update	✘	~	✓
Runs on multiple device types (e.g. laptop, tablet, smartphone, smart glass)	✘	~	✓
Runs on all operating systems	✓	✘	✓
No external software or plugins required	✘	~	✓
Technology fully developed	✓	✓	✘
Bonus requirements			
No special knowledge about telecommunication technology required	✘	✘	✓
Same programming language for all application components	✓	~	✓

Table 1: Overview of different application and architecture types and their available features

2.8 Research conclusions

The research at hand has led to the following conclusions about the prototype development of a remote support application: in order to target a maximum possible number of devices with a single code base, the prototype will be developed as a web-based application that can be accessed by browsers. With this choice, the application can be used on desktop computers with large screens as well as on smartphones or even smaller devices such as smart glasses or possibly smart watches, provided they are equipped with a web browser application.

For the connection architecture, the application uses a Node.js server for the management tasks such as user discovery and the connection establishment, and WebRTC peer-to-peer connections after finishing this process. This removes the necessity of implementing the complex logic behind video chat applications regarding network connections and security as well as video and audio streaming, since that is already part of WebRTC's default functionality and can be accessed by the JavaScript API directly in the web browser. With a client-server application implemented in Java, all telecommunication technology features would have had to be implemented autonomously or put together from various libraries, which would have introduced a considerable number of dependencies to the code base.

Another considerable advantage in the favor of WebRTC is that it can be accessed through a web browser. This makes every device that has a browser installed a potentially usable device, regardless of the operating system it is running on. Furthermore, with a Node.js web server for the management tasks, all components of the architecture use the same programming language, JavaScript. As a result, there are no interoperability issues between different languages or arduous data conversion tasks.

To implement vision enhancement, mixed reality is used. With HTML5 video and canvas elements, it is possible to render helping indicators on top the video feed without any complex logic behind it. Consequently, no computer vision algorithms are needed to recognise objects in the workspace to perform enhancement with Augmented Reality, thus reducing the technical complexity of the prototype application considerably.

Gesture recognition will not be supported in this implementation of the prototype, as this would require a substantial amount of additional hardware and rather complex computer vision algorithms. The overlay indicators should, however, provide sufficient support in terms of visual enhancement as the necessary hand movements can also be explained verbally over the audio stream.

2.9 Personal opinion

Another factor that contributed towards the deciscion of WebRTC is the author's personal preference of the JavaScript programming language. A considerable amount of knowledge and experience was gained in the past through the implementation of several applications using this language. Another particular convenience in its favor is the fact that since the introduction of Node.js, it is possible that web-based applications can be developed using JavaScript for the backend and the frontend alike, which was not possible before.

After the analysis of the criteria and requirements of remote support applications and possible ways to implement them, WebRTC is now discussed as a possible technology in detail.

3 WebRTC

Loreto & Romano (2014, p. vii) describe WebRTC as a „new standard that lets browsers communicate in real time using a peer-to-peer architecture“. This is especially interesting because it enables developers to build real-time audio and video streaming applications without any external plugins or software.

This chapter gives an overview of WebRTC, its API components and the essential elements necessary to set up a peer-to-peer connection. The history of WebRTC is discussed as well as advantages and limitations of this technology, together with its current development status.

3.1 Overview

Loreto & Romano (2014, p. vii) imply that WebRTC joins together two technologically related fields, which have been considered separately in the past: web development and telecommunication applications. This might stem from the multitude of facets that have to be considered in the telecommunication industry, while web development is a more self-contained environment.

3.1.1 Architecture

The architecture behind WebRTC is displayed below in Figure 4. On top of it stands the Web API, which is written in JavaScript and can be accessed through any web browser that has WebRTC integrated. This is the only part that developers have to keep in mind when implementing a WebRTC application. Underneath the top layer, there is a C++ API that handles the communication between the Web API and the low-level layers. The following layer is responsible for handling session management. The session layer is abstracted by design, facilitating the use of an arbitrary protocol, provided it implements the necessary methods. Furthermore, this layer handles the call cylce and management.

Located beneath the session management layer, there are the engines for audio and video streaming. They include the codecs for the recorded tracks and several methods to improve the quality of the media, such as echo cancellation and image enhancement. In this part of the architecture, the network transport mechanisms are implemented, like the Secure Real-Time Transport Protocol (SRTP) stack and the logic for settung up peer-to-peer connections, which will be explained in more detail in this chapter. In the lowest level of the architecture, all low-level tasks are carried out, like physical network transport and capturing the audio and video tracks.

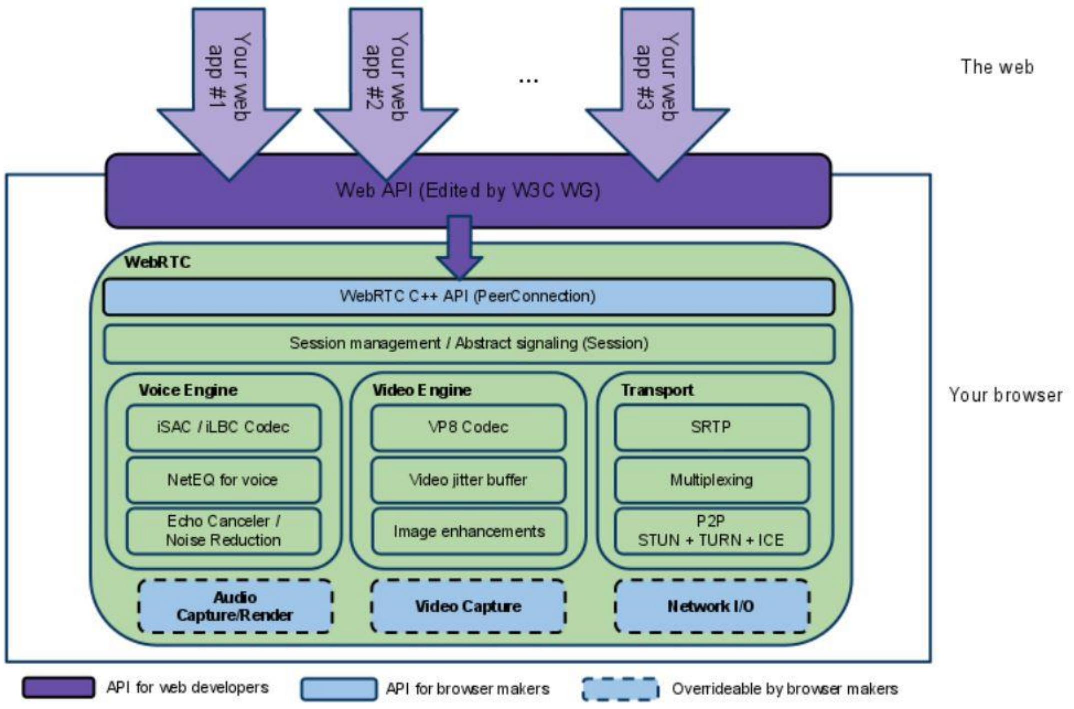


Figure 4: Overall WebRTC architecture (WebRTC Architecture n.d.)

3.1.2 Functionality and features

WebRTC enables users to establish secure audio and video streams to other peers, directly in a web browser. This is achieved without the use of external software or plugins, which was not possible before the arrival of WebRTC. It is important to note that the connections are established peer-to-peer, meaning that there are no third-party servers involved in the data traffic, only in order to set the connection up. This reduces network latency and provides an additional layer of security (cf. Azevedo et al. 2015).

In WebRTC, the well-known client-server model of the Internet is extended with peer-to-peer connections between browsers. Loreto & Romano (2014, p. 2) describe the usual scenario „to be the one where both browsers are running the same web application, downloaded from the same web page“, depicted in Figure 5.

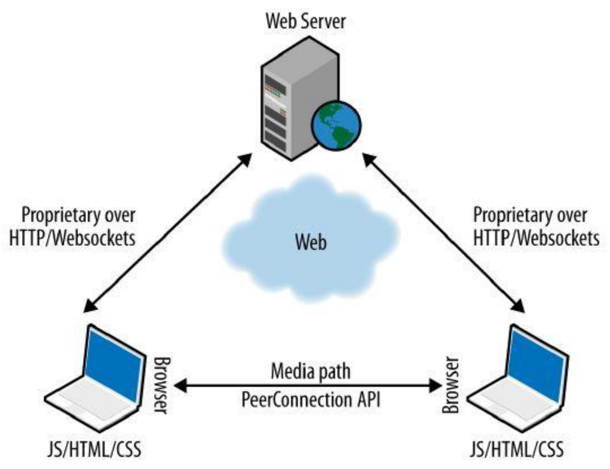


Figure 5: The WebRTC triangle (Loreto & Romano 2014, p. 3)

WebRTC web applications use web browsers as a communication interface between users. Developers implement the desired functionality using the standardised WebRTC API. This API handles the functions that are vital for a real-time communication application, such as connection management, audio and video stream access and encodings as well as data encryption (cf. Loreto & Romano 2014, p. 3f).

3.1.3 History

At the end of May 2011, Google announced – for the first time – that they were working on WebRTC and made it available to the public a couple of months later (cf. Alvestrand 2011). In November 2011 the first implementation of WebRTC was added to Google Chrome (version 23). More than a year later, at the beginning of 2013, Mozilla Firefox added their first version of WebRTC, although at that stage it only supported the *navigator.getUserMedia* function (cf. WebRTC Tutorial 2014).

An important milestone was reached in February 2013, when it was possible to use WebRTC across browser borders together on Google Chrome and Mozilla Firefox for the first time. The first mobile browsers supported WebRTC over summer and fall of 2013, when first Google Chrome for Android and a short time afterwards Mozilla Firefox for Android added their first implementations of the technology (cf. WebRTC Tutorial 2014).

3.1.4 Advantages

As Grégoire (2015, p. 1) points out, WebRTC comes with the substantial advantage of platform independence. There is no need for any external plugins to be installed, it can be used directly in a web browser. This is convenient for developers, since they do not have to ensure that their applications run on various operating systems, like Windows, Linux or Mac OS. On top of that, browser applications will also automatically work on most handheld devices, although some features from desktop computers might only be available to a limited extent, and the web page might need a responsive design to exploit its full potential.

Furthermore, encryption is mandatory for all WebRTC components, including the signaling necessary to initiate a peer-to-peer connection. As a result, the entire data transfer of WebRTC can be considered securely encrypted. For media streams, SRTP is used, for other data the Datagram Transport Layer Security (DTLS) protocol is used. Both protocols are standardised and commonly used (cf. Leaver, Iwase & Katsura 2015).

Another interesting issue worth discussing is WebRTC's media model. It is designed in a way that developers do not need any knowledge about audio and video codecs. The Internet Engineering Task Force (IETF) has decided that a minimum set of audio and video codecs must be implemented by browsers to ensure a common ground between applications running on different platforns (cf. Loreto & Romano 2014, p. 6). However, in the process of setting up a peer-to-peer connection, WebRTC tries to find the best-fitting codec that both parties‘ browser has implemented or access to. Consequently, this could be a different, non-mandatory audio or video codec.

By design, WebRTC supports two audio codecs, OPUS and G.711. This decision was made with regard to the fact that these are free of royalties. However, most common telecommunication systems use different codecs, for instance Adaptive Multi-Rate (AMR) or Adaptive Multi-Rate Wideband (AMR-WB) for mobile devices, therefore the possibility of creating an interoperability application is severely aggravated, which could lead to the addition of more mandatory codecs in the future (cf. Bertin et al. 2013)

For a long time, it was undecided which video codec would be mandatory to implement. It was a choice between the VP8 codec developed by Google or the more commonly used H.264. Similar to the audio codec situation, VP8 is free of royalties, while H.264 is more widely used by standard telecommunication applications, for instance by Skype (cf. Bertic et al. 2013). Finally, in November

2014, an agreement was reached that both VP8 and H.264 would be mandatory video codecs in WebRTC (cf. Levent-Levi 2014).

3.1.5 Limitations

A significant limitation to WebRTC at the moment is that it is still under development. Although there is a valid World Wide Web Consortium (W3C) standard, it is still possible that some functions might be added, changed or removed. As a result, many browsers still use their own vendor prefixes for methods. For instance, the RTCPeerConnection in the W3C standard is called „webkitRTCPeerConnection“ in Google Chrome and „mozRTCPeerConnection“ in Mozilla Firefox. This introduces additional sources of errors for developers. However, Google maintains the open source library adapter.js that helps programmers fix this problem. A more detailed explanation to adapter.js can be found in chapter 4.2.4.

Another interesting issue is that not all web browsers support WebRTC yet. For now, only Google Chrome, Mozilla Firefox and Opera are able to establish interoperable WebRTC connections. As shown in Figure 4 below, these three browsers represented roughly 58% of the market share in Austria in 2014. Although this is a promising quota that will likely rise further in the near future, it cannot yet be expected that an arbitrary device is capable of using WebRTC.

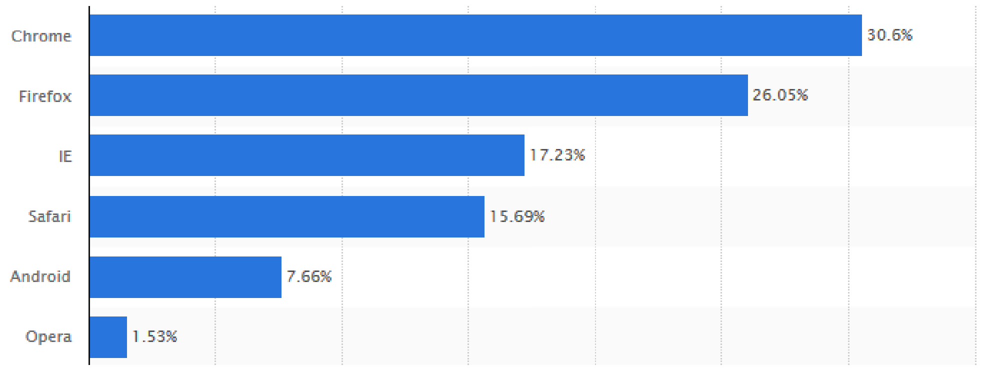


Figure 6: Web browser market share in Austria in 2014 (Statista 2015)

It has to be noted that Microsoft’s new web browser, Edge, is missing in this chart. In October 2015, it was possible to set up a peer-to-peer connection between Microsoft Edge and other WebRTC-capable browsers for the first time. In its current state, however, Microsoft Edge is not able to open DataChannel connections and the mandatory video codec implementations are also not supported yet (cf. Hancke 2015).

According to Grégoire (2015), the fact that WebRTC is running natively in web browsers does not only have advantages but also downsides. A substantial limitation is the restricted access to storage media on the device through the browser. A remedy to this disadvantage could be the use of the browser's local storage, a feature that was introduced in HTML5 (cf. Ranganathan & Sicking 2015). Additionally, a new File API is currently under development that will enable web browsers to access the local file system from the web browser, in accordance with a thorough security concept (cf. Hickson 2015).

3.1.6 Current status

For now, according to What's next for WebRTC? 2015, around 720 companies use WebRTC in some way in their products. Popular applications like Google Hangouts use it for creating DataChannels between users in chats and secure session description mechanisms (cf. Hancke 2014).

Currently, the developers behind WebRTC at Google are working on implementing the new version 1.2 of the encryption protocol DTLS and other improvements in terms of video and audio quality enhancement on mobile devices (cf. What's next for WebRTC? 2015).

3.2 API components

WebRTC consists of three main components, which developers have to implement and connect together in order to have a smoothly working application. These components are called MediaStream, PeerConnection and DataChannel. The functionality and details of all three will be explained in the following section.

3.2.1 MediaStream

To broadcast audio and video streams over the Internet, *MediaStream* objects are used. They enable the developer to interact with the streams, like displaying it in the browser window, taking snapshots or sending it to other users (cf. Loreto & Romano 2014, p. 6).

Before using a MediaStream object, it is necessary to get access to a media stream from a local media-capture device. This could be a camera or a microphone from a laptop or a smartphone. Developers can request access to these *LocalMediaStreams* by using the function *navigator.getUserMedia*. It is possible to specify the type of LocalMediaStream to be requested, audio, video or both. This is done in a configuration object that can be passed upon object initialization (cf. Loreto & Romano 2014, p. 6).

In JavaScript, the access to local media-capture devices is handled via opt-in approval from the user. When developers call the navigator.getUserMedia function for the first time, a pop-up window asks the users if they want to grant the application access to the specified media-capture devices. This approval can be revoked by both users and developers at any time so the application no longer has access to the camera or microphone.

In December 2015, Google Chrome removed the possibility to use navigator.getUserMedia on web pages that do not support Hypertext Transfer Protocol Secure (HTTPS). With HTTPS, all data transfer around the web page connection is encrypted with Transport Layer Security (TLS), thus ensuring that the data is not transferred in plain text (cf. What’s next for WebRTC? 2015). This acts as an additional layer of security, because developers are actively encouraged to use encryption in all parts of their applications.

3.2.2 PeerConnection

Instances of *PeerConnections* allow users to communicate with each other peer-to-peer, i.e. directly from one browser to another, without any web servers involved. It has to be noted, however, that a web server is always necessary for setting up a PeerConnection between two users, in order for them to find each other. This normally happens when both users visit the same web page, running on a web server which handles the peer connection setup between users. Typically, the coordination of the connection setup is handled with XMLHttpRequests or WebSockets (cf. Loreto & Romano 2014, p. 7).

3.2.3 DataChannel

While the two previous components were mandatory for a successful WebRTC connection, the third one, *DataChannel*, is optional. It offers the possibility to send arbitrary data between users connected via a PeerConnection. The DataChannel API was modeled after the WebSocket API, with similar function calls. Like WebSockets, DataChannels also offer a bidirectional connection. Developers can open an unlimited number of DataChannels within one PeerConnection, as long as each DataChannel is specified with a unique name (cf. Loreto & Romano 2014, p. 8f).

3.3 Connection setup

This section addresses the necessary parts for setting up peer-to-peer connections as well as appertaining noteworthy comments.

3.3.1 Signaling

In the WebRTC design process, it was decided to „fully specify how to control the media plane, while leaving the signaling plane as much as possible to the application layer“ (Loreto & Romano 2014, p. 5). As a result, developers do not need to handle components like video and audio formats and encodings. They do, however, have to implement the signaling in order to set up a successful WebRTC connection themselves. In practice, this means that they have to use the right API methods in the right order (cf. Loreto & Romano 2014, p. 5).

3.3.2 NAT problem

Initially, Internet Protocol (IP) version 4 (Ipv4) was used to deliver network packets over the Internet from one host to another. It uses 32-bit addresses, thus limiting the number of possible hosts to 2^{32}, or 4 294 967 296. Due to the constantly increasing demand of new Internet-capable devices and applications, one popular method to avoid IPv4 address space exhaustion was the introduction of Network Address Translation (NAT). NAT enables networks to map multiple hosts inside a network to use one public IP address, therefore reducing the usage of public IPv4 addresses.

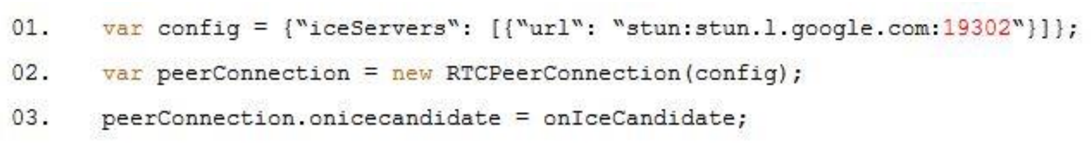
For WebRTC, however, it is vital to know the public IP address of the participating parties in order to set up a peer-to-peer connection between them. This functionality is achieved by the use of the Session Traversal Utilities for Network Address Translation (STUN) protocol. It enables an application to detect the usage of NAT in a host’s network, and to retrieve the allocated IP address and port if that is the case. A third-party STUN server is necessary in order to obtain the host’s public IP address (cf. Loreto & Romano 2014, p. 8). There are publicly available STUN servers for developers to use in this case, for example from Google (cf. Dutton 2012).

Additionally, the Traversal Using Relays around Network Address Translation (TURN) protocol extends the functionality of STUN by allowing a host inside a network that uses NAT to receive a public IP address from a relay server (cf. Loreto & Romano 2014, p. 8). As a result, the host is able to „receive media from any peer that can send packets to the public Internet“ (Loreto & Romano 2014, p. 8).

3.3.3 ICE candidates

As mentioned before, WebRTC uses PeerConnection objects to establish connections between two users. To do so, it uses the Interactive Connectivity Establishment (ICE) protocol. It facilitates peers to detect information about their network’s topology to find one or more connection paths between them, by using a variety of network protocols (cf. Loreto & Romano 2014, p. 117). Dutton (2012) points out that ICE starts with the User Datagram Protocol (UDP) first, as it has the lowest network latency. In the case that the UDP connection attempt remains unsuccessful, the Transmission Control Protocol (TCP) is used, HTTP and lastly HTTPS.

In the WebRTC JavaScript API, it is necessary to set a valid ICE server Uniform Resource Locator (URL), called the ICE Agent, in a configuration object whenever a new PeerConnection object is created (cf. Loreto & Romano 2014, p. 117). In Listing 1, a minimal example for doing so is presented, using the URL of the publicly available server from Google (cf. Dutton 2012).



```
var config = {“iceServers“: [{“url“: “stun:stun.l.google.com:19302“}]};
var peerConnection = new RTCPeerConnection(config);
peerConnection.onicecandidate = onIceCandidate;
```

Listing 1: PeerConnection ICE server config

As illustrated in Figure 7, each time a new ICE candidate is found, the ICE Agent updates the PeerConnection object and calls its *onicecandidate* callback function, in the example above *onIceCandidate* (cf. Loreto & Romano 2014, p. 117). Subsequently, the candidate is sent to the other peer. On the side of this other peer, the same process is performed, until all candidates are available to both peers. The best available candidate peer connection according to the internal WebRTC implementation is then chosen from the candidate pool. For this ICE candidate negotiation process, a server is always needed. Its sole purpose, however, is to relay the ICE candidate messages from one peer to another. After the peer connection has been established, no servers are needed for the data transfer between the parties.

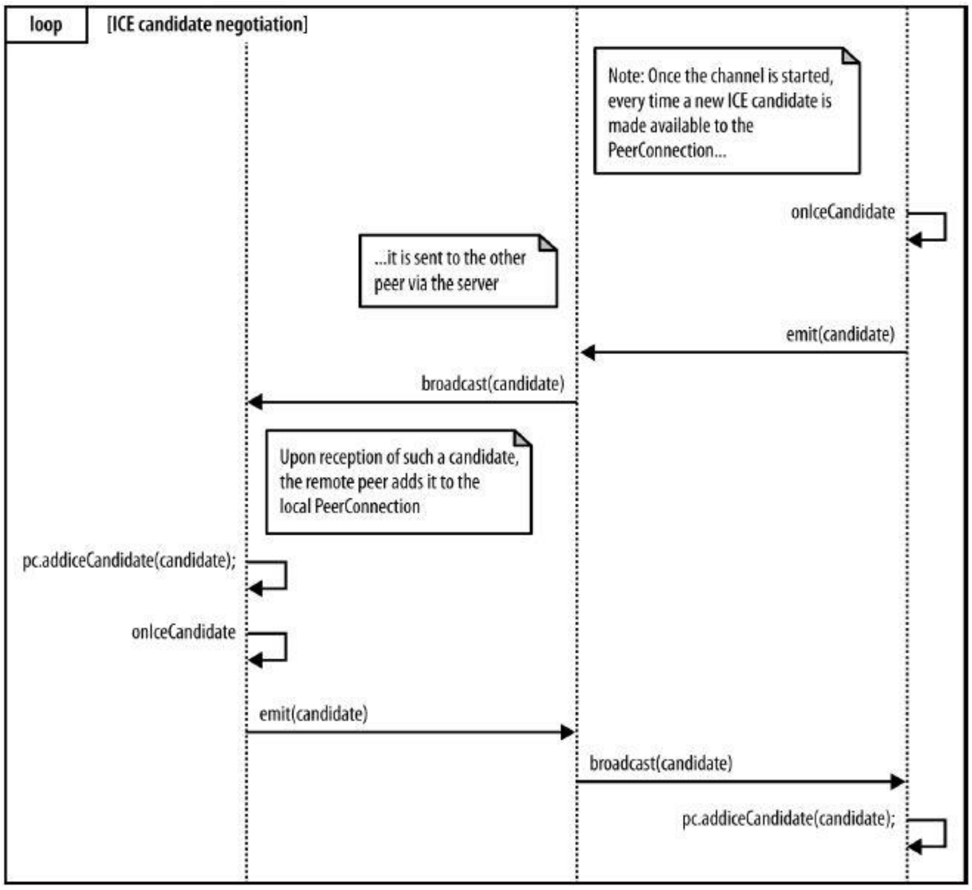

loop
[ICE candidate negotiation]
Note: Once the channel is started, every time a new ICE candidate is made available to the PeerConnection...
onIceCandidate
...it is sent to the other peer via the server
emit(candidate)
broadcast(candidate)
Upon reception of such a candidate, the remote peer adds it to the local PeerConnection
pc.addiceCandidate(candidate);
onIceCandidate
emit(candidate)
broadcast(candidate)
pc.addiceCandidate(candidate);

Figure 7: ICE candidate negotiation process (Loreto & Romano 2014, p. 118)

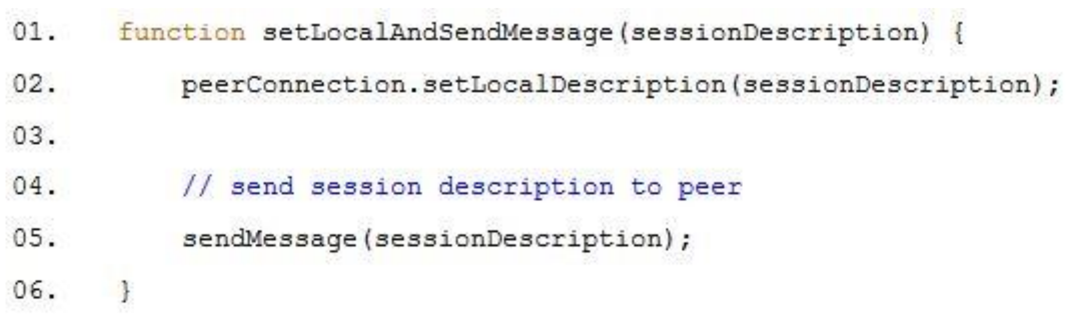
3.3.4 Session description offers and answers

It is is customary in telecommunication applications that one user calls another user. In the WebRTC architecture, this is accomplished by creating a PeerConnection object and, consequently, creating an offer and sending it to the user who is being called. The offer is created by calling the *createOffer* method on the PeerConnection object (cf. Dutton 2012).

```
peerConnection.createOffer(setLocalAndSendMessage, onSignalingError,
mediaConstraints);
```

Listing 2: Create peer offer

The parameters of the *createOffer* method are a success callback handler, an error callback handler as well as possible constraints regarding the media encoding and quality. One simple implementation of the success callback handler is described in Listing 3. The session description object is stored locally in the PeerConnection object and, additionally, serialised and sent to the remote peer (cf. Dutton 2012).

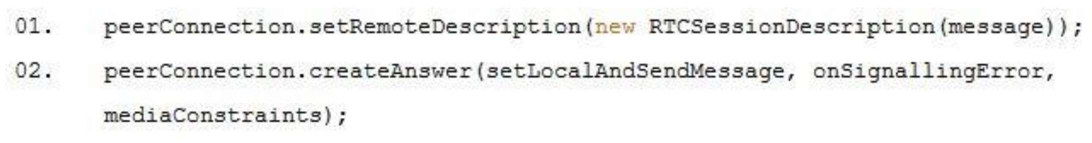


```
function setLocalAndSendMessage(sessionDescription) {
    peerConnection.setLocalDescription(sessionDescription);

    // send session description to peer
    sendMessage(sessionDescription);
}
```

Listing 3: Process session description

Upon arrival of the session description message, the remote peer registers it in its PeerConnection object. Similar to the process of creating the offer, the remote peer answers by calling the *createAnswer* method. It takes the same three parameters as the *createOffer* method (cf. Loreto & Romano 2014, p. 122).



```
peerConnection.setRemoteDescription(new RTCSessionDescription(message));
peerConnection.createAnswer(setLocalAndSendMessage, onSignallingError,
mediaConstraints);
```

Listing 4: Handling of the session description received from the peer

Now both users have exchanged session descriptions and details on how they can be located over the Internet. They are directly connected, thus no longer needing the management server to communicate with each other (cf. Loreto & Romano 2014, p. 122).

3.3.5 Data channels

Up to now, it is only possible for two users to communicate via audio and video streams. To extend this to the possibility to send arbitrary data from peer-to-peer, a DataChannel is needed. Sending data in this way, directly, without third-party servers involved, comes with substantial benefits, thanks to the low network latency (c.f. Dutton 2012). Dutton (2012) further points out that there are many use cases for this functionality, like remote desktop applications, file transfer, gaming or real-time text chat.

The API methods of DataChannels were deliberately modeled based on those from WebSockets, therefore most web developers should be fairly familiar with the syntax. Additionally, the DataChannel API brings more advantages to applications, like the usage of multiple channels within one PeerConnection, mandatory, automatic encryption as well as the support of reliable and unreliable message delivery (cf. Dutton 2012).

One user, most commonly the one creating the PeerConnection, also creates a DataChannel. There can be an unlimited number of DataChannels within one PeerConnection, identified by unique names. After the creation, three event handlers are attached to the DataChannel, which are called each time this event fires (cf. Loreto & Romano 2014, p. 125).

```
var sendChannel = peerConnection.createDataChannel(“sendChannel“, {“reliable“: true});
sendChannel.onopen = handleSendChannelStateChange;
sendChannel.onmessage = handleDataChannelMessage;
sendChannel.onclose = handleSendChannelStateChange;
```

Listing 5: Data channel setup

Because DataChannels are bidirectional, the other user does not have to create one himself. In contrast, he only has to bind an *ondatachannel* event handler to the PeerConnection object, which in turn attaches the same three event handlers to this receive DataChannel (cf. Loreto & Romano 2014, p. 125ff).

```
peerConnection.ondatachannel = gotReceiveChannel;

function gotReceiveChannel(event) {
    receiveChannel = event.channel;
    receiveChannel.onmessage = handleDataChannelMessage;
    receiveChannel.onopen = handleReceiveChannelStateChange;
    receiveChannel.onclose = handleReceiveChannelStateChange;
}
```

Listing 6: Handling of a received data channel

It is important to note, however, that unlike MediaStream and PeerConnection, DataChannel is optional to a WebRTC connection and does not necessarily have to be implemented by the developer if it is not needed.

In conclusion, this chapter examined the architecture and functionality behind WebRTC, as well as its advantages and limitations. The three core API components were described in detail, followed by the required elements regarding connection handling with additional practical considerations. In the following chapter, the process of developing a remote support application using WebRTC will be explained.

4 Prototype

With the insights and findings of the research, a prototype application was developed. The application consists of two core parts: The management server and the web interface. A vital component of the application is the overlay indicator feature, which is part of the web interface. These three most important components of the prototype are discussed in detail in the following section.

4.1 Management server

The management server is the main component of the prototype application. It performs the following tasks: First, it serves the web page and related static files, like JavaScript source files and CSS. Second, it manages WebSockets for full-duplex communication to each browser client which connected to it. Third, it carries out management and control tasks in order to set up peer-to-peer connections between users. The implementation and tasks of the management server will be described in more detail below.

4.1.1 Implementation

The management server uses Node.js[9] as a platform. Node.js is a JavaScript runtime environment that uses a single threaded, non-blocking input/output model which fits in well with JavaScript’s event looping and support for callback functions. It brings the significant advantage of using the same programming language, on the backend and the frontend. As a result, a considerable amount of programming code can be used for both parts.

4.1.2 Web server

The management server provides the web page, which has the role of the main user interface of the application, as well as all static source files. This includes JavaScript and CSS files of the application and the two external JavaScript source files, jQuery and adapter.js, which are further described in Chapter 4.2.4.

4.1.3 WebSockets

When a user opens the web page and enters a user name, a secure WebSocket connection to the management server is established. As a consequence, both parties then share a full-duplex connection and are able to exchange data at any

[9] https://nodejs.org/en/

time. This WebSocket connection stays alive until the user decides to leave the web page.

4.1.4 Management and control tasks

Each time a new user connects to the web page via WebSocket for the first time, the management server assigns a unique ID to the user. This ID is used as identification when control messages are sent to users. On the management server, there are two types of control messages: First, there are server messages. These serve the sole purpose of administration tasks, like assigning user ids or broadcasting available users, which happens every time a new user connects to the web page. Second, there are relay messages, which are used to set up peer-to-peer connections between users.

4.2 Web interface

The web interface enables users to interact with each other. Currently, users only need to provide a user name to use the application, no password is required. This is due to the fact that it is only a prototype for now. In a production environment, however, this would raise serious security issues. To eradicate this problem, a session-based login mechanism with user names and passwords could be added.

4.2.1 HTML5 elements

Where possible, modern HTML5 elements were used on the web page. This includes header, footer and section elements for the main page structure as well as video and canvas elements to display the video feeds and enable users to draw overlay indicators without the use of external plugins.

4.2.2 User interaction

The complete programming logic and user interaction was implemented with JavaScript. This includes all WebRTC functionality and the signaling communication with the management server necessary to set up connections between two peers. These functions are triggered by page interactions of the users, such as clicks on buttons.

4.2.3 Responsive design

As discussed in Chapter 2.1, remote support applications are most commonly used on mobile devices, like smartphones, tablets or smart glasses. Accordingly, an important focus in the development process was the possibility to use the application on mobile devices. It was decided to use CSS media queries to achieve a valuable user experience for all device types and sizes. Currently, there are five size breakpoints in the main CSS file, which could be easily extended to support a larger number of different screen sizes if necessary.

4.2.4 Facilitating libraries

One important principle for the prototype development was to use as few external libraries and plugins as possible. However, it was economically reasonable to use some external code to reduce the programming effort in certain areas. In the end, only two external JavaScript libraries were used: jQuery and adapter.js.

jQuery[10] was used because it is currently a de-facto standard in web development, thanks to its essential features for HTML document manipulation, event handling and useful Asynchronous JavaScript And XML (AJAX) functions. Additionally, it enables developers to write code for all common web browsers, without needing to worry about syntax differences between them.

adapter.js[11] is an open source library maintained by Google that helps developers abstract browser prefixes and API changes in the WebRTC specification. The use of adapter.js significantly reduces the amount of code necessary to implement WebRTC functionality in multiple browsers. This is due to the fact that WebRTC is currently still under development, and therefore each browser uses different function prefixes. Additionally, some API functions might be added, renamed or removed in the future. Through the use of adapter.js, developers do not need to regularly check if these parts have changed (cf. Loreto & Romano 2014, p. 96).

4.3 Overlay indicators feature

An essential feature of the prototype application is the overlay indicators feature. It enables two users sharing a peer-to-peer connection, to assist each other through drawings on the other user's screen. For now, it is possible to track the user's movements on the canvas with the mouse or with the finger or a stylus on handheld devices.

4.3.1 Implementation

For the implementation of this feature two canvases on top of each other are used. One canvas displays the current frame of the video element, which is bound to the

[10] http://jquery.com/
[11] https://github.com/webrtc/adapter/blob/master/adapter.js

media stream of the WebRTC connection. This first canvas is updated 24 times per second, appearing as a constant video stream to the user’s eye. The second canvas lies exactly on top of the first one, and is used to display the overlay indicators of the remote user. The reason for using two canvases is that the first canvas must be cleared each time it displays the current frame of the video stream and, consequently, the drawn path would also be erased. Therefore, it is necessary to use two canvases for the implementation of this feature.

4.3.2 Mouse events

The difficult part of this feature was to track the user’s movement with the mouse on an HTML canvas element. Cabanier et al. (2015) showed how geometric figures can be drawn on a canvas. As displayed in Listing 7, all that needs to be done is to draw a line from the previous touch point to the current touch point.

```
function drawPath() {
    var canvas = document.getElementById(“drawing-canvas“);
    var context = canvas.getContext(“2d“);
    context.beginPath();
    context.moveTo(previousX, previousY);
    context.lineTo(currentX, currentY);
    context.stroke();
    context.closePath();
}
```

Listing 7: Method to draw a path on an HTML5 canvas element

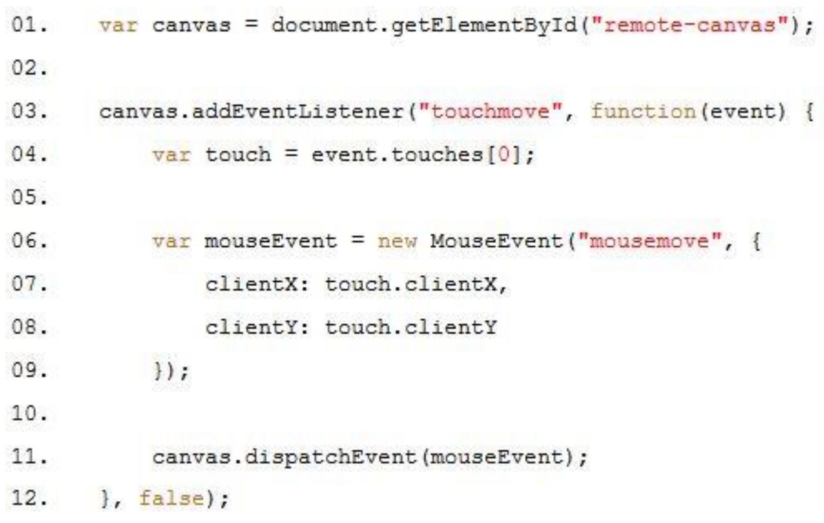
In JavaScript, it is possible to add event listeners to Document Object Model (DOM) elements, for instance when the mouse moves over the element, when a mouse button is clicked on the element or when the mouse leaves the element. In order to achieve the desired drawing functionality, it is important to not only track the movement of the mouse over the element, but also to track the state whether the left mouse button is currently being clicked or not. If that is the case, the code snippet from above gets executed each time the *mousemove* event handler is called and the path of the mouse is drawn on the canvas.

4.3.3 Touch events

While the implementation of the support drawing feature with mouse events was not tremendously difficult, the solution did not immediately work on handheld devices such as smartphones or tablets. This is due to the fact that they are operated with fingers and styluses instead of a mouse. Some handheld devices emulate mouse events when DOM elements are touched, however, the functionality of the feature on smartphones and tablets was unpredictable and unacceptable in terms of user experience.

Eventually, a solution to this problem could be found by extending the previously described logic with touch events. Touch events are the counterparts to mouse events on desktop devices. They are, however, more complex than mouse events, because while there is always only one mouse pointer on desktop devices, „a user may touch the screen with multiple fingers at the same time“ (Jenkov 2014).

For the feature at hand, it was decided to ignore multiple, simultaneous touches because it is assumed that the majority of users will not use more than one finger at a time to draw on the screen. Using this presumption, it was possible to implement the same drawing functionality on handheld devices by adding the same event handlers for touch events (*touchstart*, *touchmove* and *touchend*). Like in the example for the *touchmove* event in Listing 8, the corresponding mouse event (*mousedown, mousemove* and *mouseup*) is dispatched with the position of the touch. As a result, no additional logic is required to ensure the same functionality for touch events.



```
var canvas = document.getElementById("remote-canvas");

canvas.addEventListener("touchmove", function(event) {
    var touch = event.touches[0];

    var mouseEvent = new MouseEvent("mousemove", {
        clientX: touch.clientX,
        clientY: touch.clientY
    });

    canvas.dispatchEvent(mouseEvent);
}, false);
```

Listing 8: Dispatching a mousemove event for a touch event

4.3.4 Data transfer

To transfer the drawing path from one user to another, the DataChannel component of WebRTC was used. As described in Chapter 3.2.3, it supports the sending of arbitrary data directly from peer to peer, which exactly fulfills the requirements for the implementation of this feature.

To send the drawing path to the remote peer, in addition to the track path method being called by the canvas event listeners, a message containing the path info is sent to the connected user. The same *drawPath* method is executed with the received information in the other user's application.

4.3.5 Text chat

In addition to the possibility to communicate via audio and video stream and send support drawings to one another, it is also possible to send text messages to the connected peer, which are displayed next to the video stream. This is especially helpful in loud environments, where it is not possible to communicate with somebody verbally. Similar to the drawing path data transfer, the text chat also uses the DataChannel to send the text messages.

This concludes the chapter about the development of the prototype application and the experiences and findings of the appertaining process. The prototype is now evaluated with regard to its performance under varying video resolution settings and different conditions of network quality.

5 Evaluation

After finishing the development of the prototype application, user tests were conducted to assess the degree to which users perceived the application as helpful under varying conditions of video resolution and network quality. This is particularly interesting since remote support applications are likely to be used while moving around, as discussed in Chapter 2.1. In large factories, it is possible that wireless network reception might change in certain areas of the facility. The user tests should provide a recommendation of a minimal value of network throughput to be available at all times in order for the application to be perceived as helpful to its users.

5.1 Setup

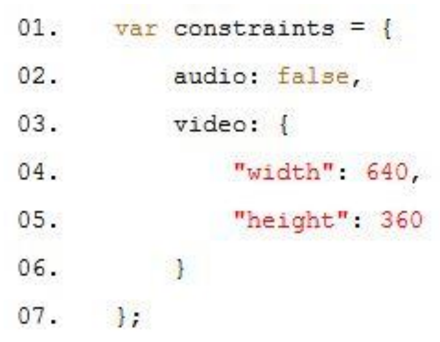
The following hardware was used to conduct the evaluation of the prototype: On the one hand, an HP EliteBook 8570p was used, running Windows 7 Professional, equipped with 8 GB RAM and a CPU clock speed of 2.60 GHz. On the other hand, a Toshiba Satellite C50 was used, running Windows 8.1 with 4 GB RAM and a 2.13 GHz CPU clock rate. The web browser in which the application was tested was on both devices Google Chrome version 48. The devices were connected over a Wireless Local Area Network (WLAN).

5.2 Method

Three different components were used for evaluating the prototype’s helpfulness to users under certain conditions. First, media constraints were used to request specific video resolutions. Second, Google Chrome Developer Tools were utilised to simulate different levels of network quality. Third, users were asked to rate their perception of the application’s usefulness under varying video and network quality. All three components are described in more detail in the following section.

5.2.1 Media constraints

Media constraints are part of the navigator.getUserMedia function described in Chapter 3.2.1. They offer the possibility of requesting certain quality standards regarding the audio and video streams. In Listing 9, for instance, a video resolution of 640 x 360 pixels is requested in the MediaStream initialization process. For more advanced setups, it is possible to add conditional settings by adding *mandatory* and *optional* objects to let the browser request an appropriate setting depending on the device’s hardware capabilities.



```
var constraints = {
    audio: false,
    video: {
        "width": 640,
        "height": 360
    }
};
```

Listing 9: A video resolution of 640 x 360 pixels is requested in the navigator.getUserMedia initialization

For the user tests, three different video resolution settings were used:

- 1280 x 720 pixels
- 640 x 360 pixels
- 320 x 180 pixels

The video frame rate in each setting was 30 frames per second.

5.2.2 Google Chrome Developer Tools

Google Chrome offers a useful tool for developers, the aptly named „Developer Tools“[12]. It provides web developers with a set of debugging tools to examine web pages regarding the time they take to load and manipulate DOM elements without having to change the source code of the application. One feature of the Developer Tools is network throttling, which simulates certain network conditions for testing purposes. There are a variety of pre-defined conditions, ranging from *offline* (no internet access) to *WiFi* (throughput of 30 Mbit/s). Additionally, developers can specify their own network conditions.

There were four different settings of network quality used for the user tests, all of which are part of the pre-defined quality simulation settings in Google Chrome:

- „GPRS“: 50 kbit/s throughput, 500 ms latency
- „Regular 3G“: 750 kbit/s throughput, 100 ms latency
- „Regular 4G“: 4 Mbit/s throughput, 20 ms latency
- „WiFi“: up to 30 Mbit/s throughput, 2 ms latency

[12] https://developer.chrome.com/devtools

5.2.3 User tests

After the technical setup, user tests were conducted. The users watched a recorded video of two people using the prototype application to solve a problem together under the previously described varying network conditions and video stream quality. Their task was to rate the degree to which they considered the application to be helpful in solving a problem, ranging from 1 (not at all helpful) to 5 (very helpful). Nine people watched the video and, subsequently, filled in an online survey of their perception of the helpfulness of the application under the present conditions. In total, there were twelve videos of thirty seconds duration shown to each user.

5.3 Results

The results of the user tests are displayed in Table 2. Users rated the application as most helpful with a video resolution of 640 x 360 under „WiFi" conditions, with a rating of 4.8. Broadly speaking, with video resolutions of 640 x 360 and 320 x 180 and network quality of „Regular 3G" or more, the application was rated 4.0 or higher by the users, thus perceived as positively helpful. Interesting to note is that in the setting with the highest video resolution of 1280 x 720, users did not perceive the application to be as helpful as with smaller resolutions. The main reason for this were significant delays of up to two seconds in the video stream and subsequently, blurred images. This was presumably caused by the additional amount of data necessary for the higher video resolution. Unsurprisingly, user also did not perceive the application as very helpful under „GPRS" conditions. Under these circumstances, the restricted network throughput led occasionally to juddering video sequences.

	Profile	GPRS	Regular 3G	Regular 4G	WiFi
	Throughput	50 kb/s	750 kb/s	4 Mb/s	up to 30 Mb/s
	Latency	500 ms	100 ms	20 ms	2 ms
Video resolution	**Frame rate**	**User helpfulness perception**			
1280 x 720	30	1.5	3.0	3.5	3.8
640 x 360	30	3.9	4.3	4.7	4.8
320 x 180	30	3.7	4.4	4.6	4.6

Table 2: Average user helpfulness perception rating from the conducted user tests

In conclusion, it is recommended to use a video resolution of 640 x 360 pixels, as under this setting, users perceived the application to be most helpful in solving a problem together over a video stream. As to be expected, the network quality should be as high as possible, but at least around 750 kb/s, as this setup received ratings of 4.0 or higher.

The following chapter discusses possible features to extend the current state of the prototype application.

6 Outlook

For now, the prototype application is not suitable for use in a production environment. Several improvements and extensions would be necessary in order to remove the current limitations of the prototype. A few suggestions for possible enhancements are therefore outlined in the following chapter.

6.1 Screenshots

A simple, though useful improvement would be the option to save screenshots from the video chat session. Additionally, these screenshots could be also sent to the other peer using the web application. This functionality could be used for the purpose of documentation or for easily assembling user guides for the repair of malfunctioning components.

6.2 User authentication

One substantial improvement to the application would be user authentication. So far, users have not been asked to provide a password, anyone was able to use the application. While this is satisfactory and in fact convenient during the development process, it is incongruous for live operation and raises severe security issues. One possibility to implement such a functionality without significant effort would be to use OAuth[13], an open standard which offers a secure authorization service that can be easily integrated into applications. With OAuth, users do not have to create a new account for using the application, but can instead use an existing account from popular web sites like Facebook or Twitter for authentication without giving away personal information about themselves.

6.3 E-mail invitations

At present, users can only call other users via the web interface. Consequently, the called user must have the web page opened in order to be notified about the incoming call. One useful extension would be the possibility to invite other users to a session by entering their e-mail address. On the management server, it would be necessary to implement some logic to generate a session ID, save it along with other meta data about the session and send an e-mail with a clickable link to the invited user that leads them directly to chat session on the web page.

[13] http://oauth.net/

6.4 Cross-platform application

So far, the prototype application has only been working in web browsers. While this offers flexibility, it would be useful to have a native app, especially for smartphones. Native app development, however, brings the disadvantage of having to implement the same application logic on multiple platforms. An economic solution to this problem would be the development of a cross-platform app, with a framework such as Apache Cordova.

6.5 Sessions with more than two users

While there can technically be an infinite number of chats running simultaneously, the number of conversational partners per chat is limited to two. This is due to the fact that WebRTC does not natively support multi-user chats. A Multipoint Control Unit (MCU) would be necessary to provide the possibility to talk to more than one person at a time. To minimise the programming effort, it would be possible to use an open-source plugin like Janus[14] that performs this task. However, this would introduce a vast number of external plugins and dependencies to the application, making it substantially more difficult to maintain and extend.

[14] https://github.com/meetecho/janus-gateway

7 Conclusion

WebRTC has received a substantial amount of attention in the technology world over the past years. It has never been easier for developers to build applications that are capable of video streaming, and now, with WebRTC, it is possible to develop them almost like any other web application. But the technology has most certainly not peaked yet. Studies predict that the number of WebRTC capable devices will rise from almost three billion in 2015 to around 6.5 billion in 2019 (see Figure 8).

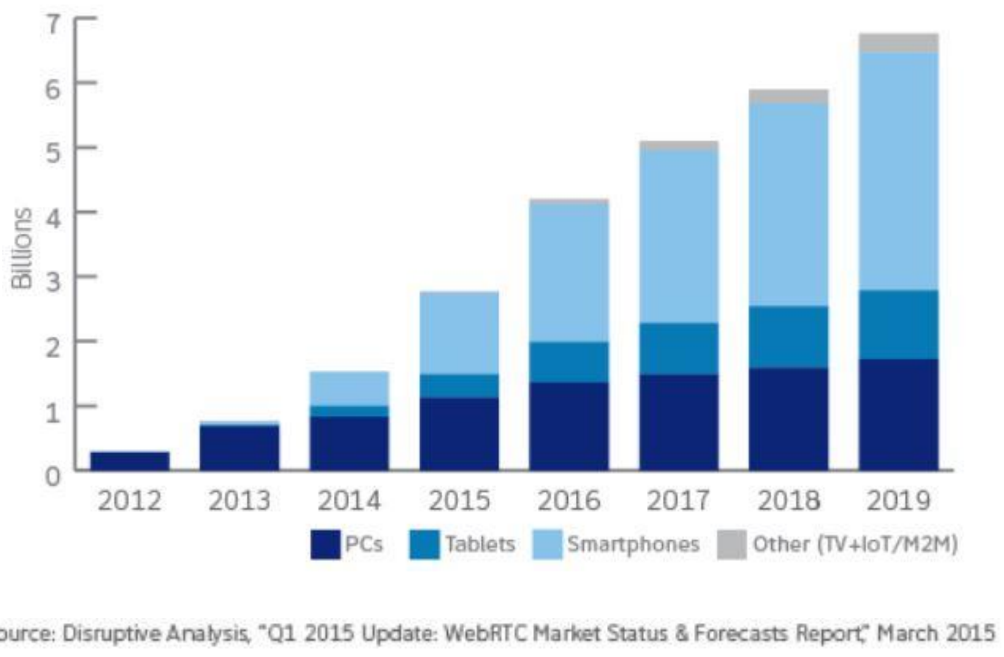


Figure 8: Expected number of WebRTC capable devices until 2019[15]

This thesis shows that WebRTC technology can be used for remote support applications. As discussed in Chapter 3.1.5, the main problem with WebRTC in its current state is, that only around 58% of mobile devices in Austria use a web browser that is capable of using WebRTC. This poses a serious problem particularly for consumer applications, where it is desired that as many devices as possible are able to use certain technologies. In business settings such as remote support applications, this problem can be considered less serious, since companies have the possibility to ensure certain technology standards within their infrastructure.

The fact that WebRTC is currently still in development might have a deterring effect for some companies to adopt it early on. However, this thesis and the developed

[15] http://webrtcstats.com/are-we-at-the-tipping-point-of-webrtc-adoption/

prototype application proved that it is possible to use this technology for helpful remote support applications even in the current development state. For the future, when a larger number of browsers support WebRTC and the internal development of the technology is finished, it will most likely be highly coveted due to the vast number of devices that can be reached with one browser based web application, including different operating systems and device types, ranging from desktop computers to wearable devices.

List of tables

Table 1: Overview of different application and architecture types and their available features 19
Table 2: Average user helpfulness perception rating from the conducted user tests 43

List of figures

Figure 1: Multiple clients, which can be of different device types, communicate with a server over the internet... 12
Figure 2: Google Glass 15
Figure 3: Apple Watch... 15
Figure 4: Overall WebRTC architecture (WebRTC Architecture n.d.) 23
Figure 5: The WebRTC triangle (Loreto & Romano 2014, p. 3) 24
Figure 6: Web browser market share in Austria in 2014 (Statista 2015) 26
Figure 7: ICE candidate negotiation process (Loreto & Romano 2014, p. 118)....... 31
Figure 8: Expected number of WebRTC capable devices until 2019 47

List of listings

Listing 1: PeerConnection ICE server config 30
Listing 2: Create peer offer 31
Listing 3: Process session description 32
Listing 4: Handling of the session description received from the peer 32
Listing 5: Data channel setup 33
Listing 6: Handling of a received data channel 33
Listing 7: Method to draw a path on an HTML5 canvas element 38
Listing 8: Dispatching a mousemove event for a touch event 39
Listing 9: A video resolution of 640 x 360 pixels is requested in the navigator.getUserMedia initialization 42

List of abbreviations

AJAX	Asynchronous JavaScript And XML
AMR	Adaptive Multi-Rate
AMR-WB	Adaptive Multi-Rate Wideband
API	Application Programming Interface
CSS	Cascading Style Sheets
CPU	Central Processing Unit
DOM	Document Object Model
DTLS	Datagram Transport Layer Security
HTTP	Hypertext Transfer Protcol
HTTPS	Hypertext Transfer Protocol Secure
ICE	Interactive Connectivity Establishment
IETF	Internet Engineering Task Force
IP	Internet Protocol
MCU	Multipoint Control Unit
NAT	Network Address Translation
RAM	Random Access Memory
RPC	Remote Procedure Calls
SRTP	Secure Real-time Transport Protocol
STUN	Session Traversal Utilities for Network Address Translation
TLS	Transport Layer Security
TURN	Traversal Using Relays around Network Address Translation
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WebRTC	Web Real-Time Communication
WLAN	Wireless Local Area Network
XML	Extensible Markup Language

Bibliography

Alvestrand H. 2011, *Google release of WebRTC source code*. Availble from: http://lists.w3.org/Archives/Public/public-webrtc/2011May/0022.html. [5 January 2016]

Azevedo J, Lopes Pereira R & Chainho P 2015, 'An API proposal for integrating Sensor Data into Web Apps and WebRTC'. In *Proceedings of the 1st Workshop on All-Web Real-Time Systems* (AWeS '15). ACM, New York, NY, USA, Article 8. ISBN: 978-1-4503-3477-8. doi: 10.1145/2749215.2749221

Bertin E, Cubaud S, Tuffin S & Cazeaux S 2013, 'WebRTC, the day after'. In *17th International Conference on Intelligence in Next Generation Networks (ICIN).* IEEE. ISBN: 978-1-4799-0980-3. doi: 10.1109/ICIN.2013.6670893

Bonsor K n.d., *How Augmented Reality Works*. Available from: http://computer.howstuffworks.com/augmented-reality.htm. [13 January 2016]

Cabanier R, Mann J, Munro J, Wiltzius T & Hickson I 2015, *HTML Canvas 2D Context,* Available from: http://www.w3.org/TR/2dcontext/. [3 January 2016]

Chen S, Chen M, Kunz A, Yantaç AE, Bergmark M, Sundin A & Fjeld M 2013, 'SEMarbeta: Mobile Sketch-Gesture-Video Remote Support for Car Drivers'. In *Proceedings of the 4th Augmented Human International Conference* (AH '13). ACM, New York, NY, USA, 69-76. ISBN: 978-1-4503-1904-1. doi: 10.1145/2459236.2459249

Ciman M, Gaggi O & Gonzo N 2014, 'Cross-Platform Mobile Development: A Study on Apps with Animations'. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing* (SAC '14). ACM, New York, NY, USA, 757-759. ISBN: 978-1-4503-2469-4. doi: 10.1145/2554850.2555104

Davies M, Zeiss J & Gabner R 2012, 'Evaluating two approaches for browser-based real-time multimedia communication'. In *Proceedings of the 10th International Conference on Advances in Mobile Computing & Multimedia* (MoMM '12), Ismail Khalil (Ed.). ACM, New York, NY, USA, 109-117. ISBN: 978-1-4503-1307-0. doi: 10.1145/2428955.2428982

Dutton, S 2012, *Getting Started with WebRTC.* Available from: http://www.html5rocks.com/en/tutorials/webrtc/basics/. [2 January 2016]

Grégoire, JC 2015, ‘On Embedded Real Time Media Communications‘*.* In *Proceedings of the 1st Workshop on All-Web Real-Time Systems* (AWeS '15). ACM, New York, NY, USA, Article 7 , 4 pages. ISBN: 978-1-4503-3477-8. doi: 10.1145/2749215.2749224

Hancke P 2014, *How does Hangouts use WebRTC? Webrtc-internals analysis*. Available from: https://webrtchacks.com/hangout-analysis-philipp-hancke/. [6 January 2016]

Hancke P 2015, *Hello Chrome and Firefox, this is Edge calling*. Available from: https://webrtchacks.com/chrome-firefox-edge-adapterjs/. [6 January 2016]

Hickson I 2015, *Web Storage*, 2nd edn. Available from: http://www.w3.org/TR/webstorage/. [5 January 2016]

Holliday MA, Houston JT & Jones EM 2008, ‘From Sockets and RMI to Web Services‘. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education* (SIGCSE '08). ACM, New York, NY, USA, 236-240. ISBN: 978-1-59593-799-5. doi: 10.1145/1352135.1352221

Huang W, Alem L, Nepal S, Thilakanathan D 2013, ‘Supporting Tele-Assistance and Tele-Monitoring in Safety-Critical Environments‘*.* In *Proceedings of the 25th Australian Computer-Human Interaction Conference: Augmentation, Application, Innovation, Collaboration* (OzCHI '13), Haifeng Shen, Ross Smith, Jeni Paay, Paul Calder, and Theodor Wyeld (Eds.). ACM, New York, NY, USA, 539-542. ISBN: ISBN: 978-1-4503-2525-7. doi: 10.1145/2541016.2541065

Jenkov J 2014, *Touch Event Handling in JavaScript.* Available from: http://tutorials.jenkov.com/responsive-mobile-friendly-web-design/touch-events-in-javascript.html. [4 January 2016]

Johansen RD, Pagani Britto TC, Cusin CA 2013, ‘CSS Browser Selector Plus: A JavaScript Library to Support Cross-browser Responsive Design‘. In *Proceedings of the 22nd International Conference on World Wide Web* (WWW '13 Companion). International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 27-30. ISBN: 978-1-4503-2038-2.

Johnson E 2015, *Choose Your Reality: Virtual, Augmented or Mixed*. Available from: http://recode.net/2015/07/27/whats-the-difference-between-virtual-augmented-and-mixed-reality/. [13 January 2016]

Leaver E, Iwase Y, Katsura K 2015, *A Study of WebRTC Security*. Available from: http://webrtc-security.github.io/. [4 January 2016]

Levent-Levi T 2014, *Who are the Winners and Losers of the WebRTC Video Codec MTI Decision?* Available from: https://bloggeek.me/winners-losers-webrtc-video-mti/. [5 January 2016]

Loreto, S & Romano SP, 2014, *Real-Time Communication with WebRTC*, 1st edn., O'Reilly Media, Sebastopol. ISBN: 978-1-449-37187-6.

Ranganathan A & Sicking J 2015, *File API*, W3C Working Draft 21 April 2015. Available from: http://www.w3.org/TR/FileAPI/. [5 January 2016]

Reinhard W, Schweitzer J, Völksen G & Weber M 1994, 'CSCW Tools: Concepts and Architectures', *Computer*, vol. 27, issue 5, pp. 28-36. doi: 10.1109/2.291293

Sinha A 1992, 'Client-Server Computing', *Communications of the ACM*, vol. 35, no. 7, pp. 77-98. *Commun. ACM* 35, 7 (July 1992), 77-98. doi: 10.1145/129902.129908

Smith J n.d., *Desktop Applications vs. Web Applications*. Available from: http://www.streetdirectory.com/travel_guide/114448/programming/desktop_applications_vs_web_applications.html. [15 January 2016]

Statista 2015, *Market share of web browsers in Austria in 2014*. Available from: http://www.statista.com/statistics/421152/wbe-browser-market-share-in-austria/. [5 January 2016]

Tehrani K & Andrew M 2014, *Wearable Technology and Wearable Devices: Everything You Need to Know*. Available from: http://www.wearabledevices.com/what-is-a-wearable-device/. [13 January 2016]

WebRTC Architecture, n.d. Available from: https://webrtc.org/architecture/. [5 January 2016]

WebRTC Tutorial, 2014 (video file). Available from: https://www.youtube.com/watch?v=5ci91dfKCyc. [5 January 2016]

What's next for WebRTC, 2015 (video file). Available from: https://www.youtube.com/watch?v=HCE3S1E5UwY. [5 January 2016]