FH JOANNEUM (University of Applied Sciences)

**Usage possibilities of WebRTC in a cross-platform developed hybrid app**

**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

06 / 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.

Michael Stifter Graz, 10.06.2016

**Table of contents**

[Abstract 5](#_Toc452231371)

[Kurzfassung 6](#_Toc452231372)

[1 Introduction 7](#_Toc452231373)

[2 Cross-platform mobile development 9](#_Toc452231374)

[2.1 Differences to native app development 9](#_Toc452231375)

[2.1.1 User experience 10](#_Toc452231376)

[2.2 Motivation 10](#_Toc452231377)

[2.3 Approaches 11](#_Toc452231378)

[2.3.1 Web apps 11](#_Toc452231379)

[2.3.2 Hybrid apps 11](#_Toc452231380)

[2.3.3 Interpreted apps 12](#_Toc452231381)

[2.3.4 Generated apps 12](#_Toc452231382)

[2.4 Cross-platform development frameworks 12](#_Toc452231383)

[2.4.1 Apache Cordova (PhoneGap) 12](#_Toc452231384)

[2.4.2 Xamarin 13](#_Toc452231385)

[2.4.3 Appcelerator Titanium 13](#_Toc452231386)

[2.4.4 Ionic 13](#_Toc452231387)

[2.4.5 Sencha Touch 14](#_Toc452231388)

[2.4.6 Other frameworks 14](#_Toc452231389)

[3 WebRTC 15](#_Toc452231390)

[3.1 History 15](#_Toc452231391)

[3.2 Architecture 15](#_Toc452231392)

[3.3 Components 16](#_Toc452231393)

[3.3.1 MediaStream 16](#_Toc452231394)

[3.3.2 PeerConnection 18](#_Toc452231395)

[3.3.3 DataChannel 19](#_Toc452231396)

[3.4 Protocols 19](#_Toc452231397)

[3.5 Functionality 20](#_Toc452231398)

[3.6 Signalling 21](#_Toc452231399)

[3.7 Potential applications 23](#_Toc452231400)

[3.7.1 Real-time communication 23](#_Toc452231401)

[3.7.2 Peer-to-peer file sharing 24](#_Toc452231402)

[3.7.3 Integrating real-time sensor data 24](#_Toc452231403)

[3.8 Advantages 25](#_Toc452231404)

[3.9 Limitations 26](#_Toc452231405)

[3.10 Usage possibilities outside of the browser 28](#_Toc452231406)

[3.11 Conclusion 28](#_Toc452231407)

[4 Prototype development 29](#_Toc452231408)

[5 Evaluation 30](#_Toc452231409)

[5.1 Setup 30](#_Toc452231410)

[5.2 Method 30](#_Toc452231411)

[5.3 Results 30](#_Toc452231412)

[6 Outlook 31](#_Toc452231413)

[6.1 User management and authentication 31](#_Toc452231414)

[6.2 Multi-user sessions 31](#_Toc452231415)

[7 Conclusion 33](#_Toc452231416)

[List of tables 34](#_Toc452231417)

[List of figures 35](#_Toc452231418)

[List of listings 36](#_Toc452231419)

[List of abbreviations 37](#_Toc452231420)

[Bibliography 38](#_Toc452231421)

Abstract

Kurzfassung

# Introduction

Over the last years, Web Real Time Communication (WebRTC) has seen a significant rise in popularity, especially in browser-based web applications. Its biggest disadvantage to date is the fact that not all web browsers support WebRTC, although the number of supporting browsers has been continuously rising for a few years now.

This poses a problem for developers who want to use WebRTC in applications today. While nowadays there are few alternatives to web applications in terms of desktop devices, the situation is different for mobile devices. Native apps have become massively popular and deliver substantial advantages when it comes to user experience. This stems from the fact that it is possible to integrate and access many components of the user’s device, such as the list of contacts, the calendar and various sensors into an application with ease. While it is possible to develop a native app that uses WebRTC, it also increases the development effort considerably, since it is necessary to implement the same functionality on multiple platforms, such as Android, iOS and Windows Phone.

A solution to this problem could be the use of a suitable cross-platform development framework that facilitates the use of WebRTC. For a cross-platform developed mobile app, it is not necessary to develop the same application once for each platform it should support, but rather only once. The framework then generates a native app from the shared code base. However, since WebRTC is a technology that can be considered relatively new and is still under development, it is not guaranteed that cross-platform development frameworks fully support the latest version of WebRTC.

This thesis takes a deeper look into popular cross-platform mobile development frameworks and examines them on their ability to support current versions of WebRTC. To analyze this examination, a set of criteria is defined in order to identify suitable frameworks for developing cross-platform apps that use WebRTC.

The thesis is structured as follows: The first part describes various ways of implementing a mobile app and highlights the advantages and disadvantages of each method in detail. The second part discusses the history and functionality of WebRTC, together with its benefits and shortcomings. In a third step, the possibilities of using WebRTC on mobile devices are addressed. Following that, the essential insights regarding the implementation of a reference app are pointed out. Chapter 5 discusses the evaluation process and its results. The final section concludes the thesis by summarizing the essential findings and suggesting possibilities to expand the underlying work.

# Cross-platform mobile development

Beginning with the introduction of Apple’s iPhone back in 2007, mobile applications have become massively popular. Back then, it was self-evident to implement an app natively on the one hand, and on the other hand there were no other options to do so. However, the following years saw a substantially increasing number of popular mobile platforms, such as Android, Windows Phone and the aforementioned iOS. Since all these platforms use different programming languages, there was no possibility to reuse the programming code written for one platform, it had to be rewritten in the exact same way for all platforms that should be supported. Additionally, making changes to an app again meant going through all platforms and implementing the changes separately for each platform (cf. PAPER-1).

APIs, MDD?

Another solution to this problem is cross-platform mobile development. It enables developers to write code for an app only once and, subsequently, generate native applications for all desired platforms from that code base. In most cases, the code is written using web technologies, such as HTML5, JavaScript and CSS. Incidentally, PAPER-2 points out, it was the original plan for apps for the iPhone to be written using these tools. In the end, however, Apple decided that third-party apps for their operating system have to be written natively in Objective-C, which was followed by Swift in 2014.

Introduction, motivation

PAPER-1

PAPER-5 [In this context, the challenge for web developers is to de-

velop di\_erent versions of their applications that are cus-

tomized to suit the speci\_c characteristics of the di\_erent

platforms, yet provide a consistent set of features and ser-

vices across all versions.]

* API

-🡪 Model Driven Development? (PAPER-6)

## Differences to native app development

A large difference between a native app and a cross-platform developed app is the fact that native apps are usually compiled, which in most cases results in faster execution times because the programming code is translated into machine code before the execution of the program. Cross-platform developed apps, on the other hand, mostly use interpreted languages such as JavaScript, which executes its code instructions step-by-step without compiling them first (cf. PAPER-2). Generated cross-platform apps, which will be discussed in more detail below, constitute an exception to this circumstance.

### User experience

One area that was heavily influenced by the spread of smartphones over the last years is user experience, a term which refers to a person’s overall experience when using a software application or system. With smartphones, for instance, it is possible that a certain part of an application is triggered when the user enters a certain location. PAPER-2 defines two primary categories for user experience: First, the context of the application or system. It consists of components which need to be understood by the user, such as platform-specific user interface conventions. Second, the implementation, which involves all elements “that can be controlled in an application” (PAPER-2). This includes the use of push notifications, the handling of input errors or the graphical layout of an application. This second part is solely the responsibility of the developer.

PAPER-2

Difference native/compiled versus web/interpreted code

Sensor/device access

User experience (push notifications, access to phonebook, contacts)

Ability to use app offline (HTML5 application cache – PAPER-3)

## Motivation

Pro / con

PAPER-7

## Approaches

PAPER-1 defines four different categories for cross-platform developed apps: Web, hybrid, interpreted and generated apps. All four approaches will be discussed in detail in this section.

PAPER-1

PAPER-9

### Web apps

Web apps are applications that run within a web browser. Typically, they use HTML5 and JavaScript. The advantage of web apps is that nowadays, almost any smart mobile device has a web browser installed, thus providing a broad range of dissemination. One disadvantage of web apps is the limited access to the device’s sensors, file system and features like contact list and calendar. Native apps, on the other hand, can exploit the device’s full potential when it comes to these features.

Unlike native apps, web apps do not need to be physically installed on the device and, furthermore, also do not have to be upgraded when a new version is available. At the same time, this becomes a disadvantage when users are not connected to the internet. In this case, the web app is not accessible to the user (cf. PAPER-1). There are modern HTML5 technologies like the Application Cache (AppCache) to eradicate this problem. AppCache allows developers to store programming logic and data on the user’s device. However, this technology requires substantial additional programming effort (cf. PAPER-3).

### Hybrid apps

Hybrid apps are a combination of native apps and web apps. They are “primarily built using HTML5 and JavaScript, and a detailed knowledge of the target platform is not required” (PAPER-1). The essential difference to web apps is that they are running within a native app container. The code is still executed by a web browser, but can be bundled together with the application, thus removing the necessity of an active internet connection to download the programming logic. With hybrid apps, it is also possible to access the device’s special features through APIs provided by the cross-platform development framework (cf. PAPER-1).

### Interpreted apps

Interpreted apps use pre-defined commands during the development process to use native user interface components. This means that on the Android platform users will interact with typical Android-styled buttons, while on iOS users will interact with iOS-styled buttons, without any effort of the developer. Despite this advantage in user experience, the developer is completely dependent on the used framework. This could especially pose a problem when a new version of an operating system is released, because it is not clear if the app will automatically have access to new features or if all previously used components will look and behave the same way (PAPER-1).

### Generated apps

This type of cross-platform developed apps use the code to generate native apps from it. They benefit from a high overall performance due to the use of compiled native code. One downside of generate apps is the increased build time that has to be carried out each time a change is made to the app (cf. PAPER-1).

## Cross-platform development frameworks

Over the last years, a multitude of cross-platform development frameworks has emerged. This section will give a brief overview of some of the most popular frameworks and mention their particular characteristics.

PAPER-4

Important criteria for choosing a framework

### Apache Cordova (PhoneGap)

One of the most popular cross-platform development frameworks is Apache Cordova[[1]](#footnote-1). Apps built with this framework belong to the category of hybrid apps. Developers can write applications with HTML5, JavaScript and CSS, which will then be executed inside a native application in a web view. Due to the fact that these tools are also used to develop web applications, this framework offers a relatively low entry point into cross-platform mobile development. Access to underlying features such as sensors and file system is provided via an API, the *Cordova Plugins*. Apache Cordova provides support for numerous platforms, such as Android, iOS, Windows Phone, Blackberry and Ubuntu.

Apache Cordova is open-source, although its owner, the software company Adobe, also released a different version of it called *PhoneGap[[2]](#footnote-2)*. PhoneGap is built on the same core application as Apache Cordova, but is part of a product package that also offers various additional tools, for instance a desktop application, a build tool and a variety of third-party libraries and plugins.

### Xamarin

Xamarin[[3]](#footnote-3) is another popular framework that builds generated native apps. Instead of web technologies it uses the programming languages C# or Ruby. Xamarin apps use native user interface components, thus providing app users with well-known interaction tools. It supports the most popular operating systems, namely Android, iOS and Windows Phone and also offers a native API to access device sensors. Xamarin also offers additional services such as an automated build tool.

### Appcelerator Titanium

Appcelerator Titanium[[4]](#footnote-4) is an example for a framework that creates interpreted apps, which means that apps created with this framework will use native user interface components. However, it also features some aspects from hybrid apps by providing developers with the possibility to write reusable modules in JavaScript.

Titanium is one product of the Appcelerator platform, together with tools like *Arrow*, which is a framework for easily building APIs or *Push*, which is a pre-built service for push notifications that can be integrated into apps. Furthermore, Appcelerator provides a multitude of analytics tools. It has to be noted that while Titanium itself is open-source and free-to-use, all other previously described tools from Appcelerator are only available in paid plans.

### Ionic

Ionic[[5]](#footnote-5) is a relatively new cross-platform mobile development framework that relies heavily modern web technologies like AngularJS[[6]](#footnote-6), Sass[[7]](#footnote-7) and virtual DOM rendering for data-intensive apps with rapidly changing user interfaces. It is built upon Apache Cordova. Ionic is an open-source project and its entire source code can be found on Github, where users are also able to report bugs or suggest improvements to the code.

### Sencha Touch

Sencha Touch[[8]](#footnote-8) focuses primarily on creating user interfaces. It has special features to simulate user interface components from Android and iOS within a web application. It does not, however, provide tools to build native apps. In order to use the code in a native app, it is either necessary to create a blank native app containing a web view for each platform the app should run on or use another framework like the previously mentioned Apache Cordova to fulfil the task.

### Other frameworks

There are also a number of smaller, lesser known cross-platform mobile development frameworks. These include jQuery Mobile[[9]](#footnote-9), Mobile Angular UI[[10]](#footnote-10), Kendo UI[[11]](#footnote-11) or the lightweight app.js[[12]](#footnote-12)

# WebRTC

“Web Real-Time Communication (WebRTC) is a new standard that lets browsers communicate in real time using a peer-to-peer architecture” (BOOK 1, p. vii). This technology development is particularly promising since it enables real-time telecommunication applications within web browsers, without the need of third-party extensions or plugins. Furthermore, WebRTC is open source software, which means that the entire source code is publicly available. This is beneficial for developers because they can get a full understanding of the inner functionality and, additionally, the code can be extended and improved by the community.

The following chapter will give an overview of the history and functionality of WebRTC, along with its benefits and limitations. The end of chapter will take a look on the possibilities of using WebRTC outside of web browsers.

## History

WebRTC started as a project conducted by Google. The first time that it was presented to the public was in May 2011. Later that year, the company decided to publish the entire source code under a permissive Berkeley Software Distribution (BSD) license, enabling the internet community to contribute ideas to the project. At the same time, in November 2011, the first rudimentary version of WebRTC was added to the Google Chrome browser. The beginning of 2013 the technology passed an important milestone, as Mozilla published its implementation of WebRTC into their Firefox browser and it was possible to start peer-to-peer connections from Chrome to Firefox and vice versa. In that same year, both companies also released the first mobile versions of their browsers supporting WebRTC (cf. WebRTC Tutorial 2014).

## Architecture

WebRTC is built upon a rather complex architecture, which can be interacted with through an API with a simple set of functions. An outline of the whole architecture is illustrated in Figure 1. On top of it stands the Web Application API, which is written in JavaScript and can be accessed in a standard web page in a browser. This is the only layer that developers have to work with, while all other architecture layers fulfil their tasks independently upon requests on this top layer. The Web Application API interacts with the internal WebRTC API, which is written in C++. The internal API is responsible for the handling of PeerConnections and their session management.

The concern of the following layer is the management of media-related components. This includes the codecs of audio and video engines, echo cancellation, image enhancement and, most importantly, the correct synchronization of media tracks in order to provide a valuable user experience. The layer below handles the capturing of audio and video streams, and is therefore directly communicating with the lowest level of the architecture, which is the physical device hardware, e.g. cameras and microphones that are built-into or attached to the device.



Figure 1: WebRTC architecture (Grigorik 2013, p. 311)

## Components

WebRTC is based upon three different components, namely MediaStream, PeerConnection and DataChannel. While the first two are mandatory, DataChannel is an additional optional component. The components are described in more detail in the following part.

### MediaStream

The MediaStream interface is responsible for everything related to streaming of audio and video. It can hold any number of MediaStreamTracks. In a traditional video conferencing scenario, this would be one video track and two audio tracks, a left and a right channel (see Figure 2). However, developers have the option to add or remove tracks (cf. BOOK-1, p. 12).



Figure 2: A WebRTC MediaStream object that contains one video and two audio tracks (BOOK-1, p. 13)

To create a MediaStream object, developers can use navigator.getUserMedia() API, which obtains access to media equipment attached to the user’s device. This typically includes cameras and microphones. Listing 1 describes a simple example on how to request access to an audio and video stream.

When this request is issued in line 21, the web browser informs the user about the request from the web page. The user can then grant access to the requested media sources or deny it. In other words, it is necessary to explicitly get the user’s approval for using the device’s media components. Furthermore, browsers display a red recording icon on top of the web page’s tab to indicate that it has currently access media resources. This adds an additional security layer on behalf of the user, since it is not possible to use the media devices without knowledge and consent from the user.

If the request was approved, the browser tries to access the requested resources and subsequently calls the success callback function described in line 8. In case the user denied the request or if an error occurred during the initialization stage, the error callback function in line 16 will be called.



Listing 1: Simple example for requesting access to camera and microphone of user device

### PeerConnection

A PeerConnection object in WebRTC “represents an association with a remote peer, which is usually another instance of the same JavaScript application running at the remote end” (BOOK-1, p. 7). In other words, it holds the actual WebRTC peer-to-peer connection between two users. It is responsible for managing every aspect of the connection, from initialization to teardown. The developer is only required to implement the initial startup and the termination of a connection, the management part in between is automatically handled by the WebRTC API (cf. PAPER-18).

The initialization of a PeerConnection is accomplished over a signalling channel, which is usually JavaScript code inside the web page. The data transfer at this stage is handled by the web server. The whole signalling process is described in more detail in Chapter 3.6. When the PeerConnection between two users has been successfully established, it is for both parties possible to exchange MediaStream objects. This could mean, for instance, that they can now see and talk to each other in a video chat directly from browser to browser (cf. BOOK-1, p. 7f).

### DataChannel

The DataChannel API is the only optional component and is therefore not required to be implemented by the developer. Its purpose is to provide an additional communications layer, in which developers can send arbitrary data between the two users. One PeerConnection object can hold any number of DataChannels. The API functions of DataChannels were modelled after the ones from WebSockets and resemble them closely (cf. PAPER-18).

The main configuration options for a DataChannel is its priority inside the PeerConnection and if the messages should be delivered in reliable or unreliable mode. In reliable mode, messages sent over the DataChannel are guaranteed to be delivered in the order they were sent, adding some administration overhead to the data transfer, which might result in slower transmission times. On the other hand, in unreliable mode this overhead is not included, thus resulting in faster transmission without guaranteeing successful delivery (cf. Ristic 2014).

## Protocols

WebRTC uses several essential protocols to deliver its real-time communications functionality. As can be seen in Figure 3, WebRTC uses User Datagram Protocol (UDP) at the transport layer, since it offers low latency and has little protocol overhead. It does, however, not guarantee the order of the packets or that a packet has been delivered at all (cf. Grigorik 2013, p. 316). In an audio and video streaming environment, this is a compromise that application designers are willing to accept, since the human brain is able to fill small gaps easily, while it is highly sensitive to transmission delays (cf. Grigorik 2013, p. 315).

With UDP alone, however, it is not possible to establish and maintain peer-to-peer connections. WebRTC needs ICE, STUN and TURN as mechanisms to determine public IP addresses and traverse NAT layers and firewalls. On top of that layer, all data transferred between two peers is secured with Datagram Transport Layer Security (DTLS). After this stage, the Stream Control Transport Protocol (SCTP) and the Secure Real-Time Transport Protocol (SRTP) handle higher-level networking tasks like multiplexing of streams and provide congestion and flow control (cf. PAPER-18). Together, all layers described in this section, provide the functionality of the PeerConnection and DataChannel API.



Figure 3: WebRTC protocol stack (PAPER-18)

## Functionality

The majority of web applications is based upon the client-server principle, which means that the client (i.e. the web browser) requests a web page from the server, who in turn fulfils the request by delivering the HTML source code of the page. In WebRTC, this model is extended by introducing a peer-to-peer communication layer (cf. BOOK-1 p. 2). As depicted in Figure 4, both peers (Alice and Bob) request a web page from a server, which also acts as signalling server. The signalling server is responsible for various tasks, such as determining the best possible option for a direct network path between the peers and finding suitable audio and video stream encodings and resolutions. After this initial signalling stage, Alice and Bob now share a peer-to-peer connection, where all media data is exchanged directly between them, without the server being involved. The media data consist of audio and video streams and, optionally, arbitrary data transferred over a DataChannel.



Figure 4: WebRTC call topology (PAPER-18)

## Signalling

An example of the signalling process is described in Figure 5. Sticking to the call setup from above, Alice is trying to call Bob. To start a peer-to-peer connection, Alice’s web application first creates a new PeerConnection object and, upon success, adds her own media stream tracks to it. This could either be audio or video tracks, or both. Afterwards, a signalling offer message is sent to the remote peer, in this case Bob. This offer message includes meta data about Alice’s media streams, such as codecs and media types, information about the network Alice is in as well as key data used to create secure connections (cf. BOOK-1, p. 9).



Figure 5: Signalling process to start a PeerConnection with another user

Bob’s application receives this offer and starts a similar process, where the same information about Bob’s endpoint is added to the PeerConnection object, which is then sent back to Alice as a signalling answer message. This process can be repeated several times, until both peers have found a suitable network path for their peer-to-peer connection. To determine this path, WebRTC uses Interactive Connectivity Management (ICE), which is responsible for locating the public Internet Protocol (IP) address of both peers. Since this might be problematic if one or both users are part of a network that uses Network Address Translation (NAT), this is accomplished by using Session Traversal Utilities for NAT (STUN) and Traversal Using Relays around NAT (TURN) servers (cf. BOOK-1 p. 8, p. 37).

The technical implementation of the signalling process can be achieved by a variety of options. One popular approach is to use WebSockets. This offers the advantage that the client and the signalling server share a persistent full-duplex connection, on which both parties can send data at any time. This is beneficial since the whole signalling process is of an asynchronous nature and requires the sending and receiving of multiple session description offers and answers. A different approach is to use XMLHttpRequest (XHR), which is as viable as the WebSockets approach from a technical point of view. For a developer, however, the use of XHR requires a more complex application architecture since it is built upon stateless, unidirectional HTTP requests (cf. Khan 2015).

## Potential applications

Due to its technical design, WebRTC excels when it comes to providing real-time peer-to-peer communication. In the near future, this will most likely open up a number of new potential use cases, especially for web applications. Three possible use cases are briefly outlined in the following.

### Real-time communication

An evident use case for WebRTC is the field of real-time communication. Before the introduction of WebRTC, developing a real-time communication application meant that programmers had to obtain vast knowledge about audio and video codecs, communication protocols, data transfer and encryption. WebRTC simplifies this process significantly by providing a plain JavaScript API (cf. PAPER-18). As a result, web developers can easily build such an application simply by calling the right API methods at the right time, without needing special knowledge about telecommunications technology.

Furthermore, due to the fact that WebRTC runs natively within web browsers, it is not necessary for the users to install any kind of software or plugin to use it. This could be an additional encouragement for people to use it, as there is no entry barrier in the form of downloading and installing software from the internet. This opens up new opportunities for e-commerce businesses to communicate with their customers personally and face-to-face, directly on their web page. For instance, these communication opportunities could include customer support or personal consulting.

In general, the field of real-time communication ranges from plain text messaging, telephone-like audio connections to video streaming. In its simplest form, there are only two users involved, but especially in a business environment, there can be a large number of people involved in a conference call (cf. PAPER-18).

### Peer-to-peer file sharing

An interesting application use case for WebRTC is peer-to-peer file sharing. PAPER-16 examined the feasibility of such an application. The idea behind it is that popular video-on-demand platforms, for example You Tube, need to invest heavily in content delivery networks (CDN) in order to provide their videos promptly to a constantly increasing number of users. PAPER-16 designed a system where all users are part of a peer-to-peer network. Theoretically, it is only necessary for one user of the network to download a video from the server. If another user requests the same file, the web application asks if an active peer already has downloaded the file. If that is the case, it gets the file directly from this user, saving a considerable amount of bandwidth for the video platform and its content delivery network.

PAPER-16 concluded that it was not possible due to the fact that web browsers at the time did not support the DataChannel API. Nowadays, however, a large number of browsers already support this API, including Mozilla Firefox, Google Chrome and Opera, making it a feasible option for platforms dealing with enormous bandwidth traffic.

### Integrating real-time sensor data

PAPER-19 propose a solution for a standardized API for the integration of real-time sensor data into web applications. They describe an example from the field of medicine, where a patient is able to remotely send data from a medical device, for instance a blood pressure sensor, to a doctor. In order for this concept to work, it is necessary to implement a middleware that handles the communication between the sensors and the web browser. This could be realized with a web browser extension (cf. PAPER-19).

Another field that could benefit greatly from such a solution are large manufactory companies. Especially since the rise of the “Industry 4.0” era, there have been several attempts to create “smart factories”, which offer increased flexibility in the production process. In such environments, malfunctions can be discovered significantly faster. They could even be detected before they happened, when the devices are able to issue warnings if measurement readings are not within pre-defined thresholds (cf. PAPER-20).

## Advantages

One significant advantage of WebRTC is the fact that it is platform independent (cf. PAPER-15). With classic desktop applications, developers had to ensure that they are functioning across a number of operating systems, such as Microsoft Windows, Linux or Apple’s Mac OS. On the one hand, this is a considerable benefit for web applications in general, since they are running in web browsers. On the other hand, however, not all web browsers offer the same range of features.

The fact that the main execution environment of WebRTC is a web browser brings the additional advantage that there are several device types that have browsers installed, which further broadens the number of platforms where web applications can be used on. This includes desktop computers, laptops, smartphones and lately also wearable devices like smart glasses and smart watches.

In addition, WebRTC brings the advantage that all components of the communication process are securely encrypted. This is of special interest to a large number of users, who are concerned about data privacy on the internet. WebRTC uses Datagram Transport Layer Security (DTLS) to encrypt the data transfers between two users in a PeerConnection. On top of that, the Secure Real-Time Transport Protocol (SRTP) is used together with the Stream Control Transmission Protocol (SCTP) to handle the real-time communication functionality, such as reliable delivery, flow control and multiplexing of media streams (PAPER-18).

IMAGE: WebRTC Protocol Stack from PAPER-18

To further improve the security of the whole WebRTC environment, Google Chrome in December 2015 removed the possibility to obtain access to a device’s camera and microphone via the MediaStream API if the web page was not loaded using Hypertext Transfer Protocol Secure (HTTPS) (cf. VIDEO-1). Consequently, developers who want to use this feature, are encouraged to run their applications in a more secure environment. If a web page does not use HTTPS, user inputs, such as data submitted in a form, are transferred to the web server in plain text. With applications like Wireshark[[13]](#footnote-13), it is possible for anyone in the same network to read the submitted data. When using HTTPS, on the other hand, the entire data transfer is encrypted with Transport Layer Security (TLS).

The design of the WebRTC architecture relies on a peer-to-peer model. In reality, this means that once a PeerConnection between two users has been established, there are no third-party servers involved in the data transfer. On the one hand, this reduces network latency because the peers are connected directly, and on the other hand it increases security by removing one component in the transfer that could be a potential target for attacks (cf. PAPER-19).

## Limitations

A significant limitation to WebRTC is its browser compatibility. As depicted in Figure 6, the web browsers that fully support WebRTC are Google Chrome, Mozilla Firefox, both together with its mobile counterparts, and Opera.



Figure 6: Overview of browsers that have a working implementation of WebRTC PeerConnections[[14]](#footnote-14)

As can be seen in Figure 7, these three browsers account for approximately 61% of the total web browser market share in Austria in 2015[[15]](#footnote-15). By comparison, this figure has risen by three percentage points from 58% in 2014.



Figure 7: Web browser market share in Austria in 2015

As for the other browsers, Microsoft’s Internet Explorer has been discontinued and cannot be installed on the current operating system Windows 10. As a result, the market share of Internet Explorer is expected to decline in the near future. Its successor, Microsoft Edge, has started to implement Object Real-Time Communication (ORTC), which is compatible to WebRTC in its current state. In September 2015, the first features of ORTC were integrated into Edge[[16]](#footnote-16). Apple, on the other hand, has not yet revealed any plans on integrating WebRTC into their Safari browser.

One reason for the incomplete browser compatibility is that WebRTC is still under development. It has a working draft API definition[[17]](#footnote-17), which is maintained by the World Wide Web Consortium (W3C) and is updated on a non-regular basis. The fact that the definition is not yet finished could possibly discourage developers, since some API functions and methods might be changed in the future. Additionally, some web browsers still use vendor prefixes for certain methods, which makes development difficult. For instance, the method for obtaining camera access is called navigator.getUserMedia() in the W3C specification, however, if developers want to ensure that it works across all browsers, they have to use the following lines of code in their application.



Listing 2: Necessary variable assignment to deal with vendor prefixes in web browsers

There are a number of other methods that suffer from the same issue, especially when it comes to WebRTC-specific functions, for instance RTCPeerConnection or RTCSessionDescription, which both must be assigned in the same way as described in Listing 2 for navigator.getUserMedia.

## Usage possibilities on mobile devices

The main goal of WebRTC is to bring real-time communications to the web browser (cf. BOOK-1, p. 1). That raises the question if there are other ways to use WebRTC on mobile devices.

### Web applications

### Native app

WebRTC offers the possibility to be used in native mobile apps. As described in Chapter 3.2, the internal WebRTC API is written C++, and offers an API for web applications. For native apps, however, there are no native WebRTC APIs yet. Google offers a WebRTC library for both the Android and iOS operating system. In order to use this library, the source code first must be compiled on the development system. The official WebRTC page offers step-by-step guidelines[[18]](#footnote-18) for the compilation process. It must be noted that for compiling the native Android code, it is necessary to use a machine which is running the Linux operating system. For the iOS code, a machine with Mac OS X is required, respectively.

### Native app with web views

Another option would be to use a standard native app that contains only one web view, which loads and displays a web application like a browser does. As a result, it is possible to provide users with a native app, without having to implement the application logic for the appropriate operating system.

Android features an implementation of WebRTC since version 4.4 of its operating system. However, as Hart (2015) points out, it is based on Chromium 36 and it is not guaranteed that it is up to date with the latest WebRTC version in web browsers. On iOS, on the other hand, there is no possibility to use WebRTC inside a UIWebView in a native app. This is due to the fact that it uses the Safari browser as an underlying foundation, which does not feature any implementation of WebRTC at all, as described in Chapter 3.9.

### Cross-platform developed mobile app

When it comes to using WebRTC inside a cross-platform developed mobile app, developers currently have two options.

First, there is Crosswalk[[19]](#footnote-19), which is an open source web application runtime. The Crosswalk Project is backed by technology company Intel and is built upon Apache Cordova. The substantial advantage of using Crosswalk is the fact it always uses the latest version of the Google Chrome browser for its web view. This is a considerable benefit when developing a WebRTC application, since Google Chrome is generally the first web browser to implement new WebRTC features (cf. Hart 2015). Due to the fact that Crosswalk uses Apache Cordova internally, applications for it are written using HTML5, JavaScript and CSS.

The second option is OpenWebRTC, which is a project by Swedish company Ericsson Research and describes itself as a “mobile-first WebRTC client framework for building native apps” [[20]](#footnote-20). It is built on top of the GStreamer multimedia framework. Like Crosswalk, the source code is open source and publicly available on the internet. OpenWebRTC cannot be considered a traditional cross-platform mobile development framework. It rather offers natively compiled libraries that provide the WebRTC functionality with API methods similar to the ones used in web applications in the browser. Using OpenWebRTC, developers can decide to either use it in a native or a hybrid app. When used in native apps, it offers the advantage of a compiled library, without the need to build the Google WebRTC library. In hybrid apps, the OpenWebRTC helper library provides custom web views for Android and iOS, which are derived from their parent classes, WebView in Android and WKWebView in iOS. As a result, it is possible to use WebRTC inside a web view on an iOS device.

## Conclusion

It has been established that the WebRTC technology can be used in a telecommunications environment and due to the fact that it is independent of specific operating systems is a considerable benefit. Furthermore, 100% of the data transfer is securely encrypted, which is particularly compelling to companies concerned about the privacy of their data.

However, the limitations described in this chapter still remain of serious nature, especially in a consumer environment. It is hardly feasible to coerce an end customer to switch to a certain web browser in order to use a web application. The same applies to a business context. While native mobile apps can provide a reliable solution for this problem, they also come with a significantly intensified effort, both in the development and maintenance stage.

On the whole, one potential compromise that can be considered both economic and user-friendly, is the use of cross-platform apps. While they suffer from certain detriments regarding user experience, the development effort is minimized and it is guaranteed that users will be able to use the app, on condition that they have a smartphone or tablet that is not older than five years.

# Prototype development

# Evaluation

## Setup

## Method

## Results

# Outlook

The underlying work is far from being finished. There are numerous possibilities for further extending the current project. A few possible enhancements are mentioned in this chapter.

## User management and authentication

The backend server that handles the WebRTC connection setup and distribution of information about available peers is currently not authenticating user requests. This does not mean that communication with the server is insecure, all requests to and from the server use HTTPS and are therefore encrypted with Transport Layer Security (TLS). However, there is currently no user management system implemented, which would allow users to log into the application with conventional username and password combinations. For now, anyone who knows the URL of the application is able to use it.

To eradicate this problem, it is either possible to design and implement an authentication solution from the ground up or use an existing application like for instance Passport[[21]](#footnote-21), which is an authentication middleware for Node. With Passport, it is possible to use local username and password authentication as well as authenticating users with an authorization protocol like OAuth[[22]](#footnote-22).

## Multi-user sessions

At this time the application allows for any number of parallel peer-to-peer sessions, meaning that one session cannot contain more than two users. While this entails a number of advantages previously discussed in this thesis, it might sometimes be necessary to invite more than two users to a session. Especially in remote support environments it might be beneficial to get the opinion of another expert to solve certain problems.

Due to its peer-to-peer design, WebRTC only supports two users in one session. If three or more users want to participate in a session, one solution would be to use a Multipoint Control Unit (MCU) as a central communication point which handles the routing of audio and video streams between all participating parties. There are publicly available open-source solutions like Erizo[[23]](#footnote-23) or Janus[[24]](#footnote-24) which could perform this task without requiring considerable development efforts.

# Conclusion

List of tables

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**

List of figures

[Figure 1: WebRTC architecture (Grigorik 2013, p. 311) 16](#_Toc452231422)

[Figure 2: A WebRTC MediaStream object that contains one video and two audio tracks (BOOK-1, p. 13) 17](#_Toc452231423)

[Figure 3: WebRTC protocol stack (PAPER-18) 20](#_Toc452231424)

[Figure 4: WebRTC call topology (PAPER-18) 21](#_Toc452231425)

[Figure 5: Signalling process to start a PeerConnection with another user 22](#_Toc452231426)

[Figure 6: Overview of browsers that have a working implementation of WebRTC PeerConnections 26](#_Toc452231427)

[Figure 7: Web browser market share in Austria in 2015 27](#_Toc452231428)

List of listings

[Listing 1: Simple example for requesting access to camera and microphone of user device 18](#_Toc452231429)

[Listing 2: Necessary variable assignment to deal with vendor prefixes 28](#_Toc452231430)

List of abbreviations

Bibliography

Grigorik I, 2013, *High Performance Browser Networking*, 1st edn., O’Reilly Media, Sebastopol. ISBN: 978-1-449-34476-4.

Hart C, 2015, *Developing mobile WebRTC hybrid applications*. Available from: <https://webrtchacks.com/webrtc-hybrid-applications/>. [29 May 2016]

Khan M, 2015. *WebRTC Signalling Concepts*. Available from: [https://www.webrtc-experiment.com/docs/WebRTC-Signalling-Concepts.html](https://www.webrtc-experiment.com/docs/WebRTC-Signaling-Concepts.html). [28 May 2016]

Levent-Levi T, 2014. *What’s Behind Ericsson’s OpenWebRTC Project?*. Available from: <https://bloggeek.me/ericssons-openwebrtc-project/>. [29 May 2016]

Ristic D, 2014. *WebRTC data channels*. Available from: <http://www.html5rocks.com/en/tutorials/webrtc/datachannels/>. [28 May 2016]

WebRTC Tutorial, 2014 (video file). Available from: <https://www.youtube.com/watch?v=5ci91dfKCyc>. [27 May 2016]

1. <https://cordova.apache.org/> [↑](#footnote-ref-1)
2. <http://phonegap.com/> [↑](#footnote-ref-2)
3. <https://www.xamarin.com/> [↑](#footnote-ref-3)
4. <http://www.appcelerator.com/titanium/titanium-sdk/> [↑](#footnote-ref-4)
5. <http://ionicframework.com/> [↑](#footnote-ref-5)
6. <https://angularjs.org/> [↑](#footnote-ref-6)
7. <http://sass-lang.com/> [↑](#footnote-ref-7)
8. <https://www.sencha.com/products/touch/#overview> [↑](#footnote-ref-8)
9. <https://jquerymobile.com/> [↑](#footnote-ref-9)
10. <http://mobileangularui.com/> [↑](#footnote-ref-10)
11. <http://www.telerik.com/kendo-ui> [↑](#footnote-ref-11)
12. <http://code.kik.com/app/2/index.html> [↑](#footnote-ref-12)
13. <https://www.wireshark.org/> [↑](#footnote-ref-13)
14. <http://caniuse.com/#search=webrtc> [27 May 2016] [↑](#footnote-ref-14)
15. <http://www.statista.com/statistics/421152/wbe-browser-market-share-in-austria/> [27 May, 2016] [↑](#footnote-ref-15)
16. <https://blogs.windows.com/msedgedev/2015/09/18/ortc-api-is-now-available-in-microsoft-edge/> [27 May 2016] [↑](#footnote-ref-16)
17. <https://www.w3.org/TR/webrtc/> [↑](#footnote-ref-17)
18. <https://webrtc.org/native-code/> [↑](#footnote-ref-18)
19. <https://crosswalk-project.org/> [↑](#footnote-ref-19)
20. <http://www.openwebrtc.org/> [↑](#footnote-ref-20)
21. <http://passportjs.org/> [↑](#footnote-ref-21)
22. <http://oauth.net/> [↑](#footnote-ref-22)
23. <https://github.com/ging/licode/tree/master/erizo> [↑](#footnote-ref-23)
24. <https://janus.conf.meetecho.com/> [↑](#footnote-ref-24)