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:** 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.

Michael Stifter Graz, 30.01.2016

**Table of contents**

[Abstract 5](#_Toc439588165)

[1 Introduction 6](#_Toc439588166)

[2 Concept 7](#_Toc439588167)

[3 WebRTC 8](#_Toc439588168)

[3.1 Overview 8](#_Toc439588169)

[3.2 API components 8](#_Toc439588170)

[3.2.1 MediaStream 8](#_Toc439588171)

[3.2.2 PeerConnection 9](#_Toc439588172)

[3.2.3 DataChannel 9](#_Toc439588173)

[3.3 Connection setup 9](#_Toc439588174)

[3.3.1 The WebRTC triangle 9](#_Toc439588175)

[3.3.2 Signaling 10](#_Toc439588176)

[3.3.3 NAT problem 10](#_Toc439588177)

[3.3.4 ICE candidates 11](#_Toc439588178)

[3.3.5 Session description offers and answers 12](#_Toc439588179)

[3.3.6 Data channels 13](#_Toc439588180)

[4 Prototype 15](#_Toc439588181)

[4.1 Management server 15](#_Toc439588182)

[4.1.1 Implementation 15](#_Toc439588183)

[4.1.2 Web server 15](#_Toc439588184)

[4.1.3 WebSockets 15](#_Toc439588185)

[4.1.4 Management and control tasks 16](#_Toc439588186)

[4.2 Web interface 16](#_Toc439588187)

[4.2.1 HTML5 elements 16](#_Toc439588188)

[4.2.2 User interaction 16](#_Toc439588189)

[4.2.3 Responsive design 16](#_Toc439588190)

[4.2.4 Facilitating libraries 17](#_Toc439588191)

[4.3 Remote support drawing feature 17](#_Toc439588192)

[4.3.1 Implementation 17](#_Toc439588193)

[5 Evaluation 19](#_Toc439588194)

[6 Possible extensions 20](#_Toc439588195)

[6.1 Screenshots 20](#_Toc439588196)

[6.2 User authentication 20](#_Toc439588197)

[6.3 E-mail invitations 20](#_Toc439588198)

[6.4 Cross-platform application 20](#_Toc439588199)

[6.5 Sessions with more than two users 21](#_Toc439588200)

[7 Conclusion 22](#_Toc439588201)

[List of figures 23](#_Toc439588202)

[List of tables 24](#_Toc439588203)

[Bibliography 25](#_Toc439588204)

Abstract

# 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 stream. For companies, this brings the substantial advantage that disruptions can be repaired significantly quicker, without the necessity of an expert having to be physically present.

# Concept

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

## Overview

### Features

### History

### 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 programmers, 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, the Secure Real-time Transport Protocol (SRTP) is used, for other data the Datagram Transport Layer Security (DTLS) protocol is used. Both protocols are standardized and commonly used (cf. A Study of WebRTC Security 2015).

### Limitations

### Current status

## API components

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

### 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 from a laptop or a smartphone or a microphone. Developers can request access to these *LocalMediaStreams* through the function *navigator.getUserMedia()*. It is possible to specify the type of LocalMediaStream to be requested, audio, video or both. (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 November 2015, Google Chrome removed the possibility to use navigator.getUserMedia() on web pages that do not support HTTP 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. (REFERENCE?)

### 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).

### 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 (REFERENCE). 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).

## Connection setup

### The WebRTC triangle

In the WebRTC architecture, 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“. This is illustrated in the figure below (Loreto & Romano 2014, p. 3).



Figure : The WebRTC triangle

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

### 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).

### 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 232, 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 all parties in order to set up a peer-to-peer connection between them. This functionality is achieved through the use of the Session Traversal Utilities for NAT (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 NAT (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). Consequently, the host is able to „receive media from any peer that can send packets to the public Internet“ (Loreto & Romano 2014, p. 8).

### ICE candidates

As described above, 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 in order to find one or more connection paths between them, by using a variety 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). Below is a minimal example for doing so, with 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;

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).

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. The whole process is illustrated in the figure below (cf. Loreto & Romano 2014, p. 118).



Figure : ICE candidate negotiation process

### Session description offers and answers

As customary in telecommunication applications, one user calls another user. In WebRTC architecture, this is accomplished by creating a PeerConnection object and, consequently, creating an offer and send it to the user who is being called. The creation of the PeerConnection was already described above, an offer is created by calling its *createOffer* method (cf. Dutton 2012).

peerConnection.createOffer(setLocalAndSendMessage, onSignalingError, mediaConstraints);

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 below. The session description object is stored locally in the PeerConnection object and, additionally, serialized and sent to the remote peer (cf. Dutton 2012).

function setLocalAndSendMessage(sessionDescription) {

peerConnection.setLocalDescription(sessionDescription);

// send session description to peer

sendMessage(sessionDescription);

}

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, onSignalingError, mediaConstraints);

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

### 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 many benefits, thanks to the low latency and high troughput (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 after those from WebSockets, therefore most web developers should be fairly familiar with the syntax. Additionally, the DataChannel API brings more advantages for applications, like the usage of multiple prioritizable channels within one PeerConnection, mandatory, automatic encryption as well as the support of reliable and unreliable message delivery (cf. Dutton 2012).

One user – in most cases the one creating the PeerConnection – also creates a DataChannel. There can be an unlimitied number of DataChannels within one PeerConnection, identified by unique names. Afterwards, three event handlers are attached to the DataChannel, which will be 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;

Because DataChannels are bidirectional, the other user does not have create one himself. Alternatively, 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;

}

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 not needed.

# Prototype

With the insights and findings of the research and the creation of the concept, 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 remote support drawing feature, which is part of the web interface. These three most important components of the prototype will be discussed in detail in the following section. The complete source code can be found in the appendix of this thesis.

## 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 Cascading Style Sheets (CSS). Second, it manages WebSockets for full-duplex communication to each connected browser. 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.

### Implementation

It was decided to use Node.js as a platform for the management server. Node.js brings the significant advantage of using the same programming language, JavaScript, on the backend and the frontend. As a result, a considerable amount of programming code could be used for both parts.

### Web server

The web page, which has the role of the main user interface of the application, is served by the management server 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.

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

### Management and control tasks

Each time a new user connects for the first time to the web page via WebSocket, the management server assigns a unique id to the user. This id is used 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.

## 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.

### HTML5 elements

Modern HTML5 elements were used on the web page where it was possible. This includes header, footer and section elements for the main page structure as well as video and canvas elements to stream the video feeds and enable users to draw support indications without the use of external plugins.

### 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 the users‘ page interactions, like clicks on buttons.

### Responsive design

Research showed that 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.

### 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 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 AJAX functions. Additionally, it enables developers to write code for all common web browsers, without needing to worry about syntax differences between them. (REFERENCE)

adapter.js 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 lines 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). (ADDITIONAL REFERENCE?)

## Remote support drawing feature

An essential feature to the prototype application is the remote support drawing. 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 stylus on handheld devices.

### Implementation

For the implementation of this feature it was decided to use two canvases on top of each other. One canvas displays the current frame of the video element, which is bound to the media stream of the WebRTC connection. This first canvas is updated 24 times per second. The second canvas lies exactly on top of the first one, and is used to display the support drawings of the remote user. Two canvases have to be used in this case because 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.

### 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. Essentially, 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();

}

In JavaScript, it is possible to add event listeners to 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 track the state whether the left mouse button is currently being clicked or not. If that is the case, then 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.

### 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 like smartphones or tablets. This is due to the fact that they are operated with fingers and styluses instead of a mouse. Most 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*), and in them, dispatch the corresponding mouse event (*mousedown, mousemove* and *mouseup*) with the position of the touch, without any additional logic.

### 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 above, 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. On the side of this other user, the same *drawPath* method is executed with the received information.

### Text chat

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

# Evaluation

# Possible extensions

At the time of writing, 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 will be outlined in the following chapter.

## 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 user via the web application. This functionality could be used versatilely, for the purpose of documentation or for easily assembling user guides for the repair of malfunctioning components.

## User authentication

One substantial improvement to the application would be user authentication. So far, users are not required to provide a password, anyone can use the application. While this is fine and in fact convenient during the development process, it is incongruous for live operation and raises security issues. One possibility to implement such a functionality without signifcant effort would be to use OAuth, an open standard which „allows secure authorization in a simple and standard method“ (OAuth n.d.). 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.

## E-mail invitations

At present, users can only call other users via the web interace. 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.

## Cross-platform application

So far, the prototype application is only 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 like Apache Cordova. It enables developers to generate native apps from one code base, through the use of HTML, CSS and JavaScript, though some platform-specific functionalities might be limited in regard to traditional native app development. (REFERENCE)

## 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 minimize the programming effort, it would be possible to use an open-source plugin like Janus (n.d.) that performs this task, altough this would introduce a vast number of external plugins and dependencies to the application.

# Conclusion

List of figures

[Figure 1: The WebRTC triangle 10](#_Toc439527718)

[Figure 2: ICE candidate negotiation process 12](#_Toc439527719)

List of tables

Bibliography

*A Study of WebRTC Security*, 2015. Available from: < http://webrtc-security.github.io/> [4 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]

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

Grégoire, J.-C. 2015, *On Embedded Real Time Media Communications.* Available from: ACM. [4 January, 2016]

Janus n.d., Available from: <https://github.com/meetecho/janus-gateway>. [2 January, 2016]

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]

Loreto, S. & Romano S. P. 2014, *Real-Time Communication with WebRTC*, 1st edn., O’Reilly, Sebastopol.

OAuth n.d., Available from: <http://oauth.net/>. [30 December, 2015]