Peer-to-peer communication in web browsers using WebRTC

A detailed overview of WebRTC and what security and network concerns exists

Christer Jakobsson

Christer Jakobsson
Spring 2015
Bachelor’s thesis, 15 hp
Supervisor: Jerry Eriksson
External Supervisor: Karl Lindmark
Examiner: Pedher Johansson
Bachelor’s program in Computing Science, 180 hp

Abstract

The web platform has a critical gap where a native proprietary application like Skype can do something a web application can't. The intention of WebRTC is to close this gap by connecting users to each other in real-time simply by loading a web page. WebRTC is technology enabling peer-to-peer communication via browsers, plugin-free. It's still a new technology and has not yet been finally standardized, nor does it come without new privacy- and security concerns. Some say that it will change the landscape of web applications while others say it's overrated.

This thesis aims to describe WebRTC, how and why it works and goes in-depth on some of the security and privacy concerns it faces. Every intermediate part will be explained in order to understand how WebRTC functions under the hood. After reading this thesis, developers should have a general idea of how WebRTC works and what concerns you need to think about when developing with WebRTC. They should also have learned that there are JavaScript libraries that simplifies how to do this in a satisfactory fashion. The paper concludes that WebRTC will potentially change the landscape of web applications but also that it will probably not happen before it is fully standardized.

Acknowledgements

I would like to thank my better half and my daughter who have supported and put up with me during the work. I would also like to thank Jenny Österlund at Acino who introduced me to the subject WebRTC and thereby providing me with the idea of the thesis. Many thanks to my external supervisor Karl Lindmark at Acino who have been my mentor and given me help and feedback and for taking the time to read this report, and finally Anders Lundberg for giving continuous valuable feedback and discussing many the many questions that emerged during the work.

Thanks to my internal supervisor at Umeå University, Jerry Eriksson. Thank you for showing interest in the field and for all the help and feedback you have given me.

June 2015, Umeå
Christer Jakobsson

Contents

1 Introduction

Web Real-Time Communication (WebRTC) is a technology that supports browser-to-browser communication for voice calling, video chat and peer-to-peer[1] filesharing without the need of internal nor external plugins. The technology is developed by Google and was released as open-source in May, 2011 [1]. Currently Chrome, Firefox and Safari have support for WebRTC without external plugins.

WebRTC differs from the existing methods to communicate via a browser in the sense that others use a server-client model where all traffic goes through the server. This effectively means that for two clients to communicate with each other, their traffic needs to go via the server. With WebRTC the server is only used to set up the connection between the peers and their traffic goes directly to the other peer.

The success of the Internet is a factor of that its core technologies such as HTML, HTTP, TCP/IP are all open and freely implementable. There currently are no free, high quality, complete solutions available that enables peer-to-peer communication in the browser. WebRTC claims to be the package that enables this [2].

All network communication technologies suffer from security concerns and WebRTC is not an exception, and therefore it is imperative to be aware of what security and privacy issues WebRTC has in order to utilize it effectively.

This thesis will describe WebRTC in detail, how all intermediate objects and parts work to establish a peer-to-peer WebRTC connection, and ultimately go in-depth on the security- and NAT traversal problems that exist for the framework.

1.1 Purpose of the thesis

This thesis aims to increase general knowledge of WebRTC and analyse how connection problems due to NAT traversing occurs and what security and privacy concerns the technology suffers from.

To sum it up, the thesis will contain a detailed overview of the intricate parts in WebRTC which enables P2P communication and answer the following questions:

- What major security and privacy issues does WebRTC have?
  - How can user identity assertion make WebRTC more secure, and how could it be implemented?
  - VPN public IP leaks: Why does it happen and how does it affect users?
- What is the NAT traversal problem WebRTC suffers from?

[1]In this thesis peer-to-peer will be abbreviated to P2P

1.2 Method

Since this is a theoretical study, to give a detailed overview of WebRTC, lots of articles and code examples have been read and analysed in order to get a understanding of how each intricate part functions in WebRTC. For each part, reading of the API documentations and/or internet-drafts by Internet Engineering Task Force (IETF) and other various sources to get an understanding of the part have been done. To further answer the questions regarding security issues and NAT traversal, articles about internet security in general and WebRTC in particular have been read, critically analysed and studied to be able to draw conclusions on the questions.

2 WebRTC: Under the hood

While WebRTC can be used for multiple tasks, its real-time P2P multimedia communication could be seen as the primary selling point for the technology. In WebRTC a client is refeered to as a peer, and in order for two peers to communicate with each other via a web browser they first need to visit a website that can relay connection negotiation data between them. Secondly, their web browsers must agree to initiate communication and they need to know how to locate one another through the internet, bypassing security and firewall protections. Lastly they need to be transmit all data in real-time.

This chapter will start of by explaining some core technologies and protocols, and then give a detailed description of each intricate part in WebRTC that makes it function the way it does. To get access to media devices WebRTC uses a JavaScript object called getUserMedia, this object is explained in detail in Appendix A.

2.1 Session Description Protocol

Session Description Protocol (SDP) is a standard to assist in setting up streaming media connections by describing their initialisation parameters. IETF published a revised specification as an IETF proposed standard in July 2006 [3].

SDP is intended to be used for describing multimedia communication sessions, invitations and parameter negotiations between the parties. SDP does not deliver the actual media but is used for negotiation between the end points, what media type and other associated properties to be used in the stream.

2.2 Network Address Translation

Network Address Translation (NAT) is a technique that maps internal network addresses to external network addresses. It makes it possible to connect several local devices to one external (Internet) connection as shown in Figure 1. When a device on the local network sends data outside of the local network, the NAT translates the IP address to the external address, or public address. When the NAT receives packages from the external address it translates it back so that the package finds the right peer in the local network [4].

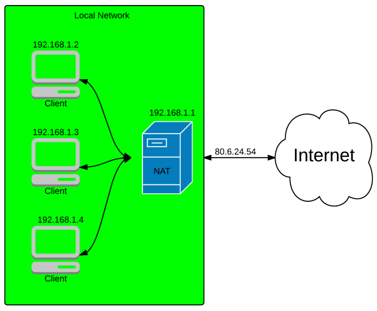

Figure 1: *Illustrates the function of a NAT*

There is a couple of different categories of a NAT setup, these are.

- *Open NAT*: Minimal restrictions or the device is compliant to Universal Plug and Play (UPnP).
- *Moderate NAT*: Minimal restrictions, but the NAT is filtering certain ports or addresses.
- *Strict NAT*: The port assignment policy is very restricted, filtering out many ports and addresses.

2.3 EventHandlers and callbacks

This section explains two fundamental concepts used in JavaScript, to enable better understanding of the code examples found in the upcoming sections.

2.3.1 EventHandler

An EventHandler is a function that detects a specific action and handles it, for example an EventHandler can be added to listen on key presses or mouse clicks. If so, when a key is pressed the EventHandler will be notified of the action and handle it. EventHandlers are often referred to as listeners in other programming languages such as Java.

2.3.2 Callback

A callback is a function that is passed as an argument to another method, and is invoked after some kind of event.

Code-example 2.1: Example of a callback

```
// Callback function
function meaningOfLife() {
    console.log("The meaning of life is: 42");
}

// Function taking a callback function
function printNumber(number, function callBackFunction()) {
    console.log("The number you provided is: "+ number);
}

// Function executing printNumber
function event() {
    printNumber(6, meaningOfLife);
}
```

When executing the `event()` function in Code-example 2.1, the result will be:

```
The number you provided is:  6
The meaning of life is:  42
```

So when `printNumber` is executed, first it will run its code and when finished, run the callback function that is provided to it, namely `meaningOfLife`.

2.4 STUN server

Network Address Translation provides a device with an IP address for use within a private local network, but this address can not be used externally and without a public address there is no way for WebRTC peers to communicate.

To get around this problem WebRTC uses Session Traversal Utilities for NAT (STUN) servers, to try and get an external address to a peer.

In a simple world, every WebRTC application would be able to learn its external address which it could exchange to other peers in order to communicate directly. In reality most devices exist behind one or more layers of NAT, firewall, proxies and anti-virus software which may block certain addresses, ports and protocols. STUN is a tool that helps protocols dealing with NAT traversal, it may be used by a device to determine the IP address and port allocated to itself by its NAT. It may also be used to check connectivity between two endpoints, and as a keep-alive protocol to maintain NAT bindings [5].

A STUN server have one simple task, to check the IP and port of an incoming request from a application that is running behind a NAT and send that address back as a response. WebRTC applications can use a STUN server to discover the <IP>:<port> from a public perspective. This enables a peer to get its own publicly accessible address and then pass it on to another peer via a signaling server in order to set up a direct link [6], see Figure 2 for an illustration.

According to webrtcstats [7] and statistics a company named Bistri has published in June 2014[1], 92% of all calls successfully made a P2P connection. That means that 8% of the traffic had to be relayed through a TURN server.

[1]Based on 1.5 million minutes of video calls done with WebRTC

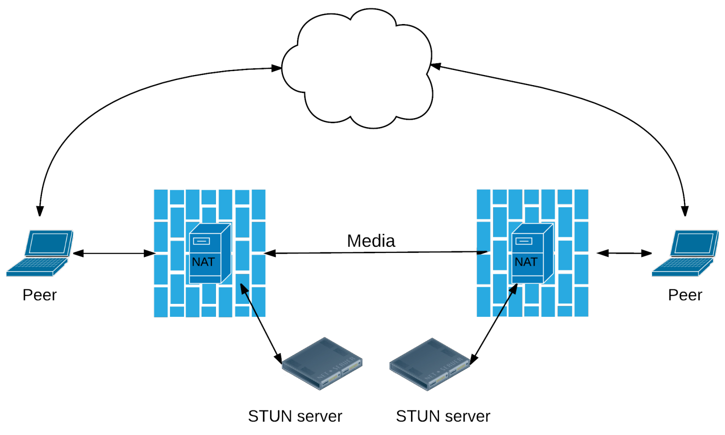

Figure 2: *Using STUN servers to get public IP:port addresses*

2.5 TURN server

Traversal Using Relays around NAT (TURN) servers is the last resort when trying to create a P2P connection with WebRTC. If a host is located behind proxies, firewalls or strict NAT's and STUN fails to get the public IP of a peer then it is impossible to set up a P2P connection. When this happens, instead of having the connection fail, WebRTC will fallback to use TURN servers to relay data between the two hosts.

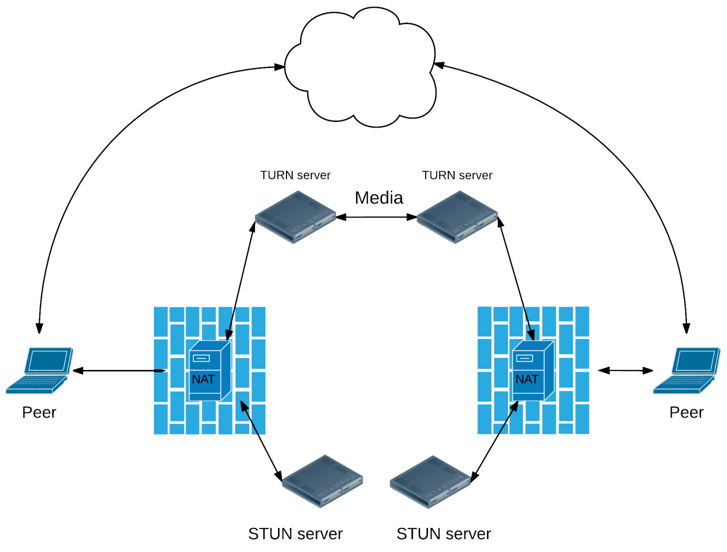

Figure 3: *This diagram shows TURN in action: pure STUN didn't succeed, so each peer resorts to using a TURN server to exchange media.*

If WebRTC needs to use a TURN server to relay the data, the communication will not be P2P but by using it as fallback WebRTC increases the odds to successfully establish connections for a wide variety of devices, as seen in the illustration shown in Figure 3.

TURN servers have a public address so they can be contacted by peers even if the peer is behind a firewall or proxy. They have a theoretically simple task, to relay data between two peers. But unlike STUN servers they will have a huge bandwidth load which results in them having to be sturdier than STUN servers. The TURN servers to be used by the application is specified in the IceConfiguration to the PeerConnection object.

2.6 RTCPeerConnection

RTCPeerConnection (simply called PeerConnection) is a component that handles multimedia communication between two peers, making sure the streams are stable and efficient. It is an API that contains functions for encryption and bandwidth management.

When setting up a connection, both peers need to initialise their PeerConnection object, as shown in Code-example 2.2. The configuration parameter is optional and it contains information to find the servers used by Interactive Connectivity Establishment (ICE), Section 2.8, [8]. There may be multiple servers of each type[2] and if no configuration is set, the default settings will be used.

Code-example 2.2: RTCPeerConnection initialisation

```
var pc = new RTCPeerConnection( configuration );
```

PeerConnection implements several functions to easier set up a connection, below follows descriptions of some:

- **createOffer(successCallback, failureCallback, optional constraints):**
  createOffer generates a SDP object that contains an offer with supported configurations for the session, including descriptions of the local MediaStream. **successCallback** is a function used when the function have executed successfully, **failureCallback** is a function that will be executed when the call fails, and the **optional constraints** parameter is used if the configuration should be changed from its default values.

- **createAnswer(successCallback, failureCallback, optional constraints):**
  createAnswer is used to respond to an offer sent from a remote connection. It generates a SDP object that contains an answer with the supported configuration for the session that is compatible in the remote configuration available in the initial offer. It contains descriptions of the local MediaStreams attached, codec options and any candidates that have been gathered by ICE. The parameters are then structured the same way as the **createOffer** function.

- **createDataChannel (DOMString label, optional dataChannelDictionary):**
  createDataChannel creates a dataChannel with the label parameter, useful for sending non video or audio data across the peer connection. The optional dictionary parameter can be used to configure the underlying channel. The function call returns a RTCDataChannel object.

[2]TURN: Section 2.5, STUN: Section 2.4

- **updateIce(optional configuration, optional MediaConstraints constraints):** updateIce updates the ICE Agent process of gathering local candidates and pinging remote ones.
- **addIceCandidate(candidate, successCallback, failureCallback):** addIceCandidate adds a remote ICE candidate to the ICE Agent. If ”IceTransports” is set to ”none” the connectivity checks will be sent to new candidates. This function call changes the state of the connection and may also result in a change of media state.

2.6.1 Offers

PeerConnection uses an offer/answer structure to mediate media information. In order to set up a multimedia communication, peers need to determine and communicate both local and remote audio and video information with each other, such as the resolution and codec capabilities. Communication to exchange media configuration goes via a signaling server and one peer sends an offer as in Code-example 2.3 with the information by using the SDP protocol.

Code-example 2.3: createOffer

```
// Get a list of friends from a server
// User selects a friend to start a peer connection with
navigator.getUserMedia({video: true}, function(stream) {
    pc.onaddstream({stream: localStream});
    // Adding a local stream won't trigger the onaddstream callback
    pc.addStream(localStream);

    pc.createOffer(function(offer) {
        pc.setLocalDescription(new RTCSessionDescription(offer), function(){
            // send the offer to a server to be forwarded
            // to the friend you're calling.
        }, error);
    }, error);
}
```

The receiver then gets the offer from the signaling server and uses createAnswer as shown in Code-example 2.4 to send a response to the caller.

Code-example 2.4: createAnswer

```
var offer = getOfferFromFriend();
navigator.getUserMedia({video: true}, function(stream) {
  pc.onaddstream({stream: stream});
  pc.addStream(stream);

  pc.setRemoteDescription(new RTCSessionDescription(offer), function() {
    pc.createAnswer(function(answer) {
      pc.setLocalDescription(new RTCSessionDescription(answer), function() {
        // send the answer to a server to be forwarded
        // back to the caller
      }, error);
   }, error);
  }, error);
}
```

In Code-example 2.3 and 2.4, `pc` is the PeerConnection object and the examples shows how to initiate a media call with video as only media.

Since WebRTC is a relatively new technology, PeerConnection has different implementations for the various browsers and to ensure that it works in Firefox, Chrome and Opera, when creating the object one needs to keep the different function calls in mind when creating the object (as seen in Code-example A.3). In Firefox it is called `mozRTCPeerConnection`, and in Chrome its either `RTCPeerConnection` or `webkitRTCPeerConnection`.

PeerConnection contains a lot of functions to modify the mediaStreams, and to handle the ICE setup, which gives it capabilities to be used in a wide variety of ways.

2.7 RTCDataChannel

RTCDataChannel (simply DataChannel) enables exchange of arbitrary data between two peers with customizable delivery properties of the underlying transport. It is a bi-directional channel and resides as a component of the PeerConnection API. Each application using the channel can configure it to provide the following:

- Reliable or partially reliable delivery of sent messages
- In-order or out-of-order delivery of sent messages

DataChannel uses a transport protocol to be able to be configurable, Stream Control Transmission Protocol (SCTP), which provides the best features of both TCP and UDP [9]. It's a message-oriented API with configurable reliablity and delivery semantics, and built-in congestion control mechanisms. TCP and UDP runs directly on top of the IP protocol but SCTP is tunneled over a Datagram Transport Layer Security (DTLS) tunnel, which runs on top of UDP.

Table 1 Comparison of TCP vs UDP vs SCTP

	TCP	UDP	SCTP
Reliability	reliable	unreliable	configurable
Delivery	ordered	unordered	configurable
Transmission	byte-oriented	message-oriented	message-oriented
Flow Control	yes	no	yes
Congestion Control	yes	no	yes

Unreliable, out-of-order delivery is equal to UDP. The message is not guaranteed to arrive at its destination, and the order it's delivered in is not important. DataChannel is also configurable to be partially reliable by specifying the maximum number of retransmissions, or setting a time limit for retransmissions. WebRTC will handle the packet acknowledgements (ACK's) and timeouts.

Reliable and in-order delivery is equal to TCP as the message is guaranteed to arrive at its destination in the same order it was sent. In Table 1 a comparison between the three protocols and their differences are shown in order to display the benefits of SCTP.

Every configuration has its own performance characteristics and limitations.

When using the channel, the `PeerConnection` object is called with: `createDataChannel( "name of channel")` to create a dataChannel with a specific name, shown in Code-example 2.5, or it is received in a channel event of the type `RTCDataChannelEvent` on an existing PeerConnection [10]. In the following examples `pc` is the PeerConnection.

Code-example 2.5: Create dataChannel

```
var pc = new RTCPeerConnection();
var dc = pc.createDataChannel("name of channel");
```

When sending data to another peer, the function `RTCDataChannel.send(data)` is used, which sends the data in the parameter over the channel. The data can be:

- *DOMString*: UTF-16 String.
- *Blob*: Represents a file-like object of immutable, raw data.
- *ArrayBuffer*: Used to represent a generic, fixed-length raw binary data buffer.
- *ArrayBufferView*: A helper type representing a set of JavaScript `TypedArray` types.

DataChannel implements EventHandlers that triggers when a message is received, the channel is open, and when the channel is closed (see Code-example 2.6 for more information). These handlers are then implemented by the application developer who uses them to create the application in the way they wants.

Code-example 2.6: EventHandlers

```
var pc = new RTCPeerConnection();
var dc = pc.createDataChannel(" name of channel");

dc.onmessage = function (event) {
    console.log("received: " + event.data);
};

dc.onopen = function () {
    console.log("datachannel open");
};

dc.onclose = function () {
    console.log("datachannel close");
};
```

DataChannel makes the most of features built into PeerConnection, for example the use of ICE to get through NAT's and it has many potential applications. Another benefit is improvement of latency in using WebRTC instead of other server-client based communication models such as Websockets. DataChannel may in the future be the most popular tool for gaming, file-transfer, real-time text chat applications on the web [11].

2.8 Interactive Connectivity Establishment

ICE is a framework used in computer networking to deal with NAT's, to establishing communication for interactive media such as Voice over IP (VoIP), instant messaging and P2P data. According to Jonathan Rosenberg [12], protocols that use the offer/answer method have difficulties traversing through NAT's and retrieve the correct IP address and port for peers.

The WebRTC ICE deployment has two endpoints, known as peers, that want to communicate. They are able to communicate indirectly via a signaling server where they can perform an offer/answer exchange of SDP messages. In the beginning of the ICE setup, peers don't know their own topology and are not aware if they are behind one or several NAT's. ICE helps the peers discover enough information about their own topology so that they potentially find one or more paths by which they can communicate with each other.

ICE will try all possible paths in parallel and try to choose the most efficient path that works and each possible path is called an ICE Candidate. First it tries to make a connection using the host IP address obtained from the device operating system and network card, which will fail if the device are behind NAT's. ICE obtains the the external address by using a STUN server and if for some reason that address also fails traffic will be relayed through a TURN server.

URLs for STUN and/or TURN servers are (optionally) specified by a WebRTC application in the iceServers configuration object which is the first argument to the PeerConnection constructor shown in Code-example 2.7. Multiple servers of each kind can be specified and ICE will try them in the order they are set in the configuration file.

Code-example 2.7: iceServers configuration

```
{
  'iceServers': [
    {
      'url': 'stun:stun.l.google.com:19302'
    },
    {
      'url': 'turn:192.158.29.39:3478?transport=udp',
      'credential': 'JZEOEt2V3Qb0y27GRntt2u2PAYA=',
      'username': '28224511:1379330808'
    },
    {
      'url': 'turn:192.158.29.39:3478?transport=tcp',
      'credential': 'JZEOEt2V3Qb0y27GRntt2u2PAYA=',
      'username': '28224511:1379330808'
    }
  ]
}
```

First ICE tries to use UDP, but if it fails it will proceed with both HTTP and HTTPS over TCP. If direct connection fails, most often because of enterprise NAT traversal issues and firewalls, ICE uses a TURN server. In other words, ICE will first use STUN with UDP to directly connect peers and, if that fails, will fall back to a TURN relay server. The expression 'finding candidates' refers to the process of finding network interfaces and ports.

2.9 Signaling server

WebRTC enables P2P-communication but it still needs servers to deal with NAT's and firewalls that complicates establishing a connection. Furthermore, signaling servers are needed to make peers exchange metadata which is used to coordinate communication [6].

In order for peers to set up a WebRTC connection, information needs to be exchanged.

- *Session control messages:* Used to open or close communication.
- *Media metadata:* Such as codecs and codec settings, bandwidth, and media types.
- *Key data:* Used to establish a secure connection.
- *Network data:* Host IP address and port as seen by the outside world.
- *Error messages*: Used to show information when something goes wrong.

The signaling process is not implemented by WebRTC's API so it is up to the software developer that wants to implement WebRTC to build it for their application. As stated by Sam Dutton [6], signaling methods and protocols are not specified by WebRTC standards and not implemented into the API to avoid redundancy and maximize compatibility with established technologies. This approach is outlined by JavaScript Session Establishment Protocol (JSEP).

JSEP is a protocol that removes the need for browsers having to save the state. If browsers would need to save the state it would make the setup problematic if signaling data would be lost each time a page was reloaded. JSEP pulls the signaling state machine out from the browser and into JavaScript. Doing this removes the browser from the signaling flow and the only interface needed is a way for the application to pass in the local and remote session descriptions negotiated by the signaling mechanism. Instead the signaling state can be saved on the server [13]. JSEP requires peers to exchange offer and answer, which contains the metadata with information about the media. An illustration of what an offer, answer can look like can be seen in Figure 4 where the caller sends an offer with the metadata in the form of a SDP. The offer goes through the signaling server and the callee is then able to answer the call or reject it. If the callee sends an answer a WebRTC connection will be established and the peers can start to exchange data.

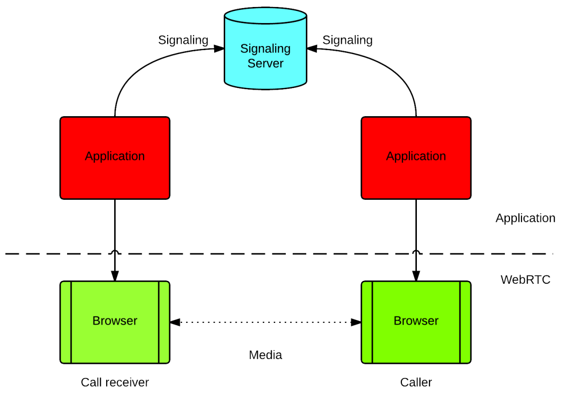

Figure 4: *JSEP Architecture, used when initiating a WebRTC connection*

According to an Internet draft from IETF [14] JSEP is simple and straightforward in its SDP-handling. When an offer/answer exchange is needed, the caller creates an offer by calling PeerConnections createOffer function. After that the application optionally modifies the offer and uses it to set up its local configuration by using the setLocalDescription function. Then the offer is sent off to the callee over the signaling mechanism of choice. When receiving the offer, the callee installs contents from the offer by using the setRemoteDescription function in PeerConnection.

To sum it up, the signaling server is used by WebRTC in order for peers to exchange information about the connection they want to establish with each other, and to negotiate its terms. A signaling server is often also used as the web server that serves the HTML, CSS and JavaScript which creates the means for peers to find others to connect to.

2.10 JavaScript Libraries

There exists a great deal of JavaScript libraries that have the intentions of making WebRTC easier to use in some context, such as.

- **Simplify**:
  - Hiding the code that sets up the connection, ex: ICE candidates
  - Making it easier to develop an application where users have multiple WebRTC connections at the same time.
- **Specialize**:
  - Focus on the ability of screen capturing with WebRTC.
  - Focus on Audio/Video recording in the browser.
  - Audio/Video processing in the browser
  - Focus on DataChannel and using it in game-development.

In this thesis the focus will be on JavaScript libraries that tries to simplify usage of WebRTC in the field that it supposedly was intended for, namely multimedia communication. Three libraries will be briefly explained and each one claims to be the fastest, easiest way to implement WebRTC video, audio and data applications. A detailed overview of how to use the library to set up a DataChannel, multimedia call, and some function calls it implements are explained in detail in Appendix A

2.10.1 PeerJS

PeerJS is a JavaScript library created by two independent developers, and is supposed to simplify the implementation of WebRTC. Basically it is an API that aims to be configurable, complete, and easy to use. It wraps the browsers WebRTC implementation so that with nothing but an ID, peers can create a P2P data or media stream in just a few simple steps [15].

PeerJS hides the difficulties of WebRTC for developers, so they don't have to worry about ICE configuration or setting of local or remote descriptions. Shown in Code-example A.4 only an API key needs to be supplied when creating the peer object when using the free cloud hosted signaling server. However, in the case where a private server is being used, the destination IP and port needs to be supplied.

2.10.2 EasyRTC

EasyRTC is a JavaScript library that consists of a peer/browser-side and a backend JavaScript server built on top of node.js. It is open-source and claims to be the fastest, most robust, easiest way to implement secure WebRTC video, audio and data applications for the enterprise [16].

Like other Javascript libraries easyRTC needs to be loaded in the HTML page to be able to use it (shown in Code-example A.15). EasyRTC is only available for download and upon being successfuly included, an `easyrtc` object that contains all methods needed to use the library is created. This object is then used for every interaction with the connection, from connecting to disconnecting. One crucial difference with easyRTC is that the library provides a built-in fail-safe back to websockets. The reasoning behind this is to make it more reliable for peers to successfully connect to other peers and if websockets are used the signaling server is used to relay the data.

2.10.3 SimpleWebRTC

SimpleWebRTC is a JavaScript library that consists of many other smaller JavaScript libraries (called modules) and the purpose of it is to simplify usage of WebRTC. To accomplish this, simpleWebRTC assumes very much about what type of web application the developer wants to build [17].

SimpleWebRTC needs to be loaded in order to be used, Code-example A.18 shows two examples of doing this; one example where the script is linked from the website, and another where it first is downloaded and linked from the local computer. The simpleWebRTC team claims that its unlikely simpleWebRTC will have the exact features wanted, and therefore it is good that it is comprised of smaller modules [17].

2.11 Summary

When two peers wants to create a multimedia connection between each other, the web-applications JavaScript will use PeerConnection with getUserMedia to get permission to use the camera/microphone devices. Each peer will contact the signaling server and exchange offer/answer to agree on the parameters for the call. Then ICE will execute its procedure by using the STUN server to get the public and local IP address of the user and get candidates, try every candidate and pick the best one as the route. If no candidate is working one of the users are behind a strict NAT/firewall and WebRTC resorts to use a TURN server to relay data. If a candidate is working, a P2P communication session will start and both peers will exchange media, illustrated in Figure 5. All this will potentially be done by using one of the many JavaScript libraries that exists for WebRTC.

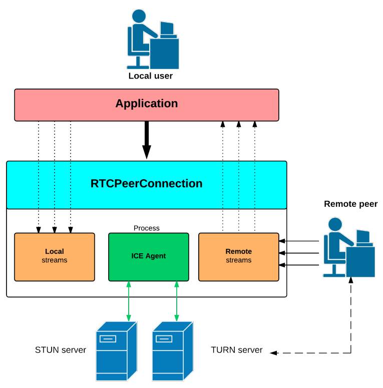

Figure 5: *Picture showing the structure of WebRTC*

3 Security and NAT traversal issues

This chapter consists of profound information about certain specific aspects concerning WebRTC, to be able to answer the questions formulated in the introduction. These questions are.

- What major security and privacy issues does WebRTC have?
  - How can user identity assertion make WebRTC more secure, and how could it be implemented?
  - VPN public IP leaks: Why does it happen and how does it affect users?
- Define the meaning of the NAT traversal problems WebRTC suffers from.

First general security concerns are explained and continues to go in-depth on each specific security- and privacy question and explain what the problem is as well as what kind of repercussions it may have. Afterwards a thorough analysis of the NAT traversal problems that WebRTC is presented.

3.1 Security issues

WebRTC is the first browser-based technology that breaks the strict server-client architecture on the web by enabling direct browser-to-browser communication without an intermediary server relaying the data. Specifications are intentionally open to allow multiple solutions by decoupling the signaling- and media planes and not specifying which protocol to be used as signaling. However, while mostly beneficial for all parties, the variety among the existing solutions ultimately means that there is no single security solution that will work for every implementation [18]. WebRTC having a large combination of utilities makes it an interesting target for a broad set of attacks. Attackers can for example:

- Abuse the JavaScript API's.
- Hijack the service provider's website.
- Create identity confusion.
- Create connections between wrong users.

Traditional two-party, client-server web security measures involves using HTTPS, but in more complex interactions such as filetransfer or VoIP applications whereas WebRTC is used and a server exists to mediate communications between the users it is not as effective.

In these cases users need to trust the server to not use the data in a negative way, logging, mine personal information or modify the messages.

WebRTC suffers from these problems because the intermediary signaling server is just a means to get to the entities they really want to communicate with. This effectively means that a malicious WebRTC site could connect to the wrong person. WebRTC can't function without signaling servers due to many reasons, for example the connectivity can change and a signaling server is then needed for the peers to re-exchange offer/answers. Therefore the security issues need to be resolved for WebRTC to be at least every bit secure as its competitors.

In order for WebRTC applications to work, each peer downloads some HTML, CSS, JavaScript from the web server. This server often also acts as a signaling server and since WebRTC relies on JavaScript code to both acquire and render media streams and create connections to other peers, the result is that potentially, malicious code is running in the users browser. The server used to relay communication is what end-to-end security mechanisms tries to control.

3.1.1 User identity assertion

The server can potentially record or reroute a users messages or calls. To achieve end-to-end security without trusting the calling site a new way to control authentication is needed and Identity Assertion Provider (idP) is therefore introduced. An idP can be used in conjunction with WebRTC to provide verification that the user is an authorized user and can also return other information about the user in question - for example username or email. This authority-based identity model also needs to be independent from the calling site to ensure that it is secure, an illustration of idP implementation for WebRTC are shown in Figure 6) [19].

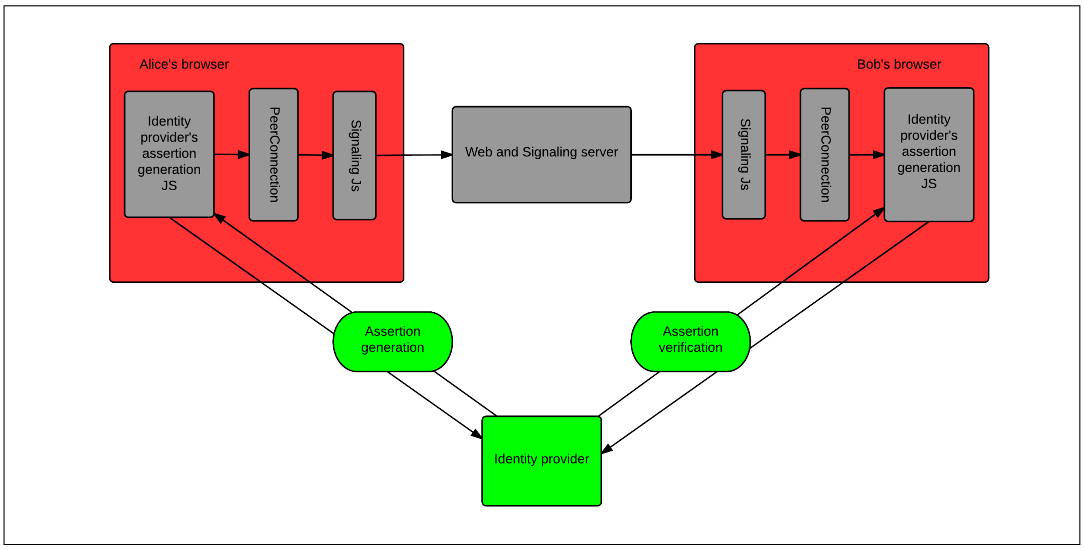

Figure 6: *Lifecycle of a WebRTC identity assertion*

When implementing an authority-based identity model, any web server in principle can act as an idP as long as it is capable to provide JavaScript for assertion generation and verification. This code verifies the user's identity and generates an identity assertion, binding the user's identity to the public key used in the Datagram Transport Layer Security - Secure Real-Time Transport Protocol (DTLS-SRTP) session [20].

When a peer wants to make a call, the WebRTC applications signaling server send an assertion to the callee. The assertion contains the domain name of the identity provider and for the callee to verify the identity the callee's browser downloads the identity providers assertion-verification code and asks to verify the assertion which it received from the signaling server.

If WebRTC would implement a way to easily use identity assertion we would be sure that a user is the same user as the last time, but we would still not know if the user is in fact a talking dog.

3.1.2 Real IP leaks when using VPN

This section will cover the problem of WebRTC enabling websites to get access to a users public IP address even when behind a VPN. The information for the section is gathered from various online sources that show a interest in security- and privacy related topics. These are, lifehacker [21], torrentfreak [22], thehackernews [23], unhappyghost [24] and privateinternetaccess [25].

A Virtual Private Network (VPN) can be used to hide the real public IP address and is used for many reasons like hiding location or to be harder to trace. VPN users are now facing a massive security flaw as websites can easily see their real public IP address with the help of a WebRTC STUN server. A few lines of code is all it takes to remove the location protection a VPN provides, and figure out exactly where a user is located and what internet service provider (ISP) it has, which in turn can tie the address to who the user specifically is.

According to Ernesto [22], the flaw is limited to supporting browsers and it supposedly only affects Windows users. However, according to Alan Henry [21] their own tests show that it is depending on your VPN and how it is configured, which means that your IP address may be leaked even if you use a Mac or Linux system.

How it works

WebRTC allows requests to be made to STUN servers which return the hidden public IP address as well as the local network address for the system that is being used by the user.

These request results are available to JavaScript, so you can now obtain a users local and public IP addresses in JavaScript. Additionally, these STUN requests are made outside of a normal XMLHttpRequest procedure, which means that they are not visible in the developer console or able to be blocked by plugins such as AdBlock or Ghostery [21].

How it can be fixed

There are many potential ways to fix the VPN leak, one way being to use a browser that does not support WebRTC such as Internet Explorer. But most often we dont want to use a different browser and in these cases there are scripts that disables WebRTC, or it can be done manually. There are differences in the way to do this in every browser and disabling WebRTC will obviously have the result that it won't be possible to use.

The best way to be protected from this flaw is to run the VPN tunnel directly on the router instead of on the local computer. This will also have several other benefits such as that VPN will be applied on Wi-Fi connected devices and that every local device will be behind the VPN; smartphones, smart TV and so forth. In other words, making the home network safer.

There are two downsides with the approach to have the VPN configured at router level, and these are.

- User wants to change exit server often, meaning: one day the user wants to browse as though he is in Japan, another in Iceland.
- User only needs to use the VPN occasionally, for example enable VPN when working and disable when not working.

Both of these downsides have the problem that the user needs to access the router and reconfigure the VPN service every time and this process can sometimes be both complicated and time consuming [21].

This VPN vulnerability is a serious flaw that compromises users as it is well hidden and a user behind a VPN might be unaware that if he or she enters a site that uses WebRTC JavaScript might log their real public IP address. The upside is that its fairly easy to resolve, the problem with these fixes is that they effectively disables WebRTC, for the user which shuts out a big chunk of potential users of a system that uses WebRTC, hence this is a serious problem that WebRTC has to overcome for it to be adopted by the big masses.

Summary

There exists security issues for WebRTC and some are issues that can be found in other fields of the web platform, such as malicious JavaScript and someone hacking the service providers website. Identity Assertion could be used to increase the security by ensuring that a user is whoever claimed to be and it is possible to be implemented alongside WebRTC. Public IP leak behind VPN is a major security issue that VPN users need to be made aware of in order to protect themselves, and the issue needs to be resolved in a better way then what is available now.

3.2 NAT Traversing problems for WebRTC

In order for a WebRTC P2P connection to be established, the two endpoints needs to be able to create a direct connection to each other. This has shown to be problematic because of NAT and firewall devices, such as home or enterprise routers, which often also acts as a NAT and firewall. Mostly the normal home router is not as restrictive as a router used by an enterprise and therefore users behind enterprise routers have the biggest problem to successfully create a WebRTC connection. A signaling server is used to find peers to connect to, and exchange information to be able to set up a session. It would be displeasing if the resulting session needs to be relayed via a TURN server for a number of reasons.

- Multimedia sessions are extremely sensitive to latency. P2P generally results in lowest possible latency.
- Direct P2P connections most often have fewer hops than relayed traffic, which results in lower chance of packet loss.
- TURN servers are commonly not distributed geographically, thus if traffic would need to be relayed it would increase the latency.

Enterprises and home networks use firewalls to enforce certain rules for access policies of IP's. These rules associate who is allowed to access which resource, and is most often incorporated in a router with NAT capabilities. Conventional firewall devices are developed to mainly handle server-client protocols such as web-browsing, e-mail and file-transfer, since this is the biggest and oldest technology used to communicate on the web. A client-server packet sent from the inside of a firewall device to the outside normally creates a passage at the firewall so that the packets sent back are not blocked. Essentially, the communication must be initiated by the host in the local network [26].

Although server-client communications works well through NAT/firewalls, it plays mischief with P2P communications. P2P normally works by endpoint applications opening up or listening with a specific IP and port combination on the system. It can then signal to another host what IP and port it is prepared to exchange data on, but when there are NAT's and firewalls in-between, the port or IP set to communicate over might not be open to allow data though it.

P2P communications have been historically difficult to get through enterprise firewalls [27] and WebRTC is no exception. It is however well equipped with a robust set of tools built into the framework, which allows web-developers to be somewhat oblivious using WebRTC.

Summary

There will always be hurdles to overcome for network technologies, and WebRTC seems to have hit a nerve that the normal client-server architecture handles with ease. Firewalls, proxies and NAT's are not going anywhere and WebRTC is equipped to handle them as best fit, there seems like most often a P2P connection can be established. The problem may be that WebRTC is a technology that will mainly be used by people behind enterprise firewalls/NAT's which often are so strict that a direct connection is not possible. NAT traversal concerns is one of the major drawbacks of WebRTC, it does contain tools like a TURN server to relay data so that a connection will almost always succeed, but this lead to the communication not being P2P, hence removing all advantages of using WebRTC. But according to webrtcstats.com [7] 92% of all traffic was P2P, proving that WebRTC, even with its NAT traversal issues is beneficial.

4 Conclusions

This chapter contains the conclusions drawn for each qustion discussed in Chapter 3, starts with general security issues. Then follows conclusions on the two sub questions, and ends with what has been concluded on the NAT traversal problem question.

4.1 Security issues

WebRTC is plagued with some seemingly big security- and privacy concerns, and at first glance a it seems risky to make use of the technology. What one needs to understand is that almost all network technology has major security problems, and in this paper the focus has remained on the disadvantages rather than advantages in these aspects. WebRTC does have positive security and privacy aspects since P2P traffic is harder to log, and the technology has encryption options.

4.1.1 Identity and Malicious JavaScript

Identity concerns is not a problem merely for WebRTC but the way it can be implemented is restricted to P2P technologies that use signaling. It has some great benefits and the thesis presents a way identity assertion could be implemented.

The problem with identity and malicious JavaScript running in the users browser is not specific to WebRTC but it exists for other web applications too, with the difference that WebRTC can't really adopt solutions from other areas since the technology is unlike other web technologies. The upside is that WebRTC's signaling is not specified, which makes it possible to integrate an idP with the signaling process.

To conclude, identity assertion could make WebRTC more secure by providing assurance that a user is the same user as last time, but still not assurance that the user is whoever the user claims to be.

4.1.2 Real IP leaks when using VPN

After reading articles about WebRTC's security issues and discovering the VPN IP leak issue, the conclusion came to be that the leak happens because of the STUN servers WebRTC use which can find out the public IP address of a client by a simple request. The issue affects users because the solution that currently exists is dependant on that the user behind a VPN is aware of the issue, and takes the time to resolve it. This can be done by configuring the users web-browser to not allow WebRTC to be run, either by installing a plugin or by disabling it in the browsers configuration. But this solution effectively shuts out users behind a VPN from using WebRTC safely.

While WebRTC isn't the only technology that uses STUN servers, it's the combination of STUN and potentially malicious JavaScript code running in the users browser that poses the threat called IP leak. And in turn makes it problematic to create a worthwhile solution, this presumably is the biggest security issue that doesn't seem to have a solution just yet.

What is the impact of the security issues?

In short, the problems with verifying a users identity results in WebRTC applications where, without an idP, the user may not be whoever he claims to be. The thesis describes a method how an idP could be implemented in the WebRTC framework to be able to assert a users identity. The VPN leak compromises users privacy, making the web unsecure for them since they have no immediate way of knowing whether or not a website is using WebRTC. This limits the potential userbase of WebRTC.

4.2 NAT Traversing problems for WebRTC

After investigating what NAT traversal is and how it affects WebRTC. The conclusion to the meaning of what NAT traversal problems WebRTC suffers from came to be:

- Strict firewalls/NATs making it difficult for a peer to learn its public IP address.
- Firewall/NAT devices primarily developed to handle server-client protocols.
- Problems to create a direct connection between two peers.

5 Discussion and future work

This chapter consists of a discussion on the sources, results and other interesting facts that is presented, and it will of finish with a section discussing potential future work.

When studying the VPN leak problem and searching for information about it, all sources came to be ones that where published in various internet community's that discuss topics of security, privacy and other computer related topics. The downside with these sources is that they have medium to low credibility, but since the information gathered from them are facts, the amount of sources gave assurance that they where correct.

The VPN IP leak issue results in a big chunk of users that either know of the problem and have disabled WebRTC, making them unable to benefit from sites that use it, or users that use WebRTC and are unaware of the leak, hence their identity is compromised. In order for WebRTC to hit the global market and be used in a global scope, where it is the method to use when designing a web application with P2P, this security issue needs to be addressed.

Each of the items that makes WebRTC work is still not finalized and they may change. This can cause it to not be ready to be fully adopted because risk of high maintenance when changes are made to the API.

The statistics in Section 2.4 that shows that 92% of all WebRTC calls are P2P can be questioned. It only refers to how many minutes of video calls that has been studied, not how many calls that have been made or that calls where done from different locations, behind different NAT/firewalls setups. This gives that the statistics may not be as good as stated.

The thesis goes through three JavaScript libraries that embeds WebRTC and aims to simplify its usage in some manner. There exists a wide range of libraries and this makes it harder to use WebRTC in the way that it is hard to know which library is best fitted for use in a specific application.

The way conclusions have been drawn could be questioned as all information is taken from the works of others and my own opinions in the subject. Although the articles that have been used are mostly credible sources, some of the online sources that concerns the VPN IP leak have low credibility. However, they do have strength in numbers as most of them say the same thing. They simply state the problem, its effect and also a possible solution to it.

WebRTC is one of the most promising changes in the HTML5 family, and it has the potential to change how we use the internet by making it more collaborative and more interactive. But there are lots of hurdles to overcome, W3C needs to get Microsoft and Apple on board and they need to ensure a consistent experience across the browsers that support WebRTC. Until then, belief is that it will remain a niched technology.

Future work

The contents of this thesis have studied WebRTC in security and privacy issues and discussed the NAT traversal problems.

A next step could be to create a JavaScript library that contains a solution that have identity assertion with WebRTC, this could benefit the community and gain more developers that is interested in the technology. Other areas to continue work could be to study a solution to the VPN IP leak problem or to analyse the JavaScript libraries and see if there is one that could evolve and be the one and only library used by the majority.

One could also continue investigating the ramafications of the issues with NAT traversing for WebRTC and if there is something that could be done to soften the issue and what lies in the future, will it get easier or harder to traverse NAT's?

Bibliography

[1] H. Alvestrand. (2011, May) Google release of WebRTC source code. Last accessed: 2015-05-13. [Online]. Available: http://lists.w3.org/Archives/Public/public-webrtc/2011May/0022.html

[2] C. J. A. N. Daniel Burnett, Adam Bergkvist. (2015, April) WebRTC.org homepage FAQ section. Last accessed: 2015-05-13. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/MediaStream_API

[3] V. J. Mark Handley. (2006, July) SDP: Session description protocol. Last accessed: 2015-05-13. [Online]. Available: http://tools.ietf.org/html/rfc4566

[4] M. H. Pyda Srisuresh. (1999, August) IP network address translator (NAT) terminology and considerations. Last accessed: 2015-05-13. [Online]. Available: http://tools.ietf.org/html/rfc2663

[5] P. M. R. M. D. W. Jonathan, Rosenberg. (2008, October) Session traversal utilities for nat (stun). Last accessed: 2015-05-13. [Online]. Available: http://tools.ietf.org/html/rfc5389

[6] S. Dutton. (2013, November) Webrtc in the real world: Stun, turn and signaling. Last accessed: 2015-05-13. [Online]. Available: http://www.html5rocks.com/en/tutorials/webrtc/infrastructure/

[7] WebRTCstats. (2014, June) WebRTC revolution in progress! Last accessed: 2015-05-13. [Online]. Available: http://webrtcstats.com/webrtc-revolution-in-progress/

[8] C. J. A. N. Daniel Burnett, Adam Bergkvist. (2015, Mars) WebRTC 1.0: Real-time Communication Between Browsers. Last accessed: 2015-05-13. [Online]. Available: http://w3c.github.io/webrtc-pc/archives/20150306/webrtc.html

[9] I. Grigorik, *High Performance Browser Networking: Chapter 18. WebRTC.* O'Reilly Media, 2013.

[10] k. t. tOkeshu, m0ppers. (2015, May) RTCDataChannel. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/RTCDataChannel

[11] S. Dutton. (2013, February) RTCDataChannel for chrome. Last accessed: 2015-05-13. [Online]. Available: http://updates.html5rocks.com/2013/02/WebRTC-data-channels-API-changes-and-Chrome-talks-to-Firefox

[12] J. Rosenberg. (2010, April) Interactive connectivity establishment (ICE). Last accessed: 2015-05-13. [Online]. Available: https://tools.ietf.org/html/rfc5245

[13] J. Uberti. (2011, December) Javascript session establishment protocol (jsep). Last accessed: 2015-04-28. [Online]. Available: http://lists.w3.org/Archives/Public/public-webrtc/2012Jan/att-0002/JavascriptSessionEstablishmentProtocol.pdf

[14] C. j. Uberti, Justin. (2015, FFebruary) draft-ietf-rtcweb-jsep-03. Last accessed: 2015-05-13. [Online]. Available: http://tools.ietf.org/html/draft-ietf-rtcweb-jsep-03#section-1.1

[15] E. Z. Michelle Bu. The peerjs library. Last accessed: 2015-05-13. [Online]. Available: http://peerjs.com/

[16] P. S. Inc. easyrtc homepage. Last accessed: 2015-04-24. [Online]. Available: http://easyrtc.com/

[17] &yet. Simplewebrtc.js from &yet. Last accessed: 2015-04-27. [Online]. Available: http://simplewebrtc.com/

[18] L. Desmet and M. Johns, "Real-time communications security on the web," *Internet Computing, IEEE*, vol. 18, no. 6, pp. 8–10, 2014.

[19] V. Beltran, E. Bertin, and N. Crespi, "User identity for WebRTC services: A matter of trust," *IEEE Internet Computing*, no. 6, pp. 18–25, 2014.

[20] R. Barnes and M. Thomson, "Browser-to-browser security assurances for WebRTC," 2014.

[21] A. Henry. (2015, October) How to see if your VPN is leaking your IP address. Last accesed: 2015-06-2.

[22] E. V. der Sar. (2015, January) Huge security flaw leaks VPN users real IP address. Last accessed: 2015-05-13. [Online]. Available: https://torrentfreak.com/huge-security-flaw-leaks-vpn-users-real-ip-addresses-150130/

[23] M. Kumar. (2015, February) WebRTC vulnerability leaks real IP addresses of VPN users. Last accessed: 2015-05-13. [Online]. Available: http://thehackernews.com/2015/02/webrtc-leaks-vpn-ip-address.html

[24] U. Goldenstein. (2015, February) WebRTC killing Tor, VPN, IP masking, privacy. Last accessed: 2015-05-13. [Online]. Available: http://www.unhappyghost.com/2015/02/webrtc-killing-tor-vpn-ip-masking-privacy.html

[25] cchen. (2015, January) How to stop WebRTC local IP address leaks on google chrome and mozilla firefox while using private ip. Last accessed: 2015-05-13. [Online]. Available: https://www.privateinternetaccess.com/forum/discussion/8204/how-to-stop-webrtc-local-ip-address-leaks-on-google-chrome-and-mozilla-firefox-while-using-private-i

[26] Reid. (2013, August) An intro to WebRTC's NAT/firewall problem. Last accessed: 2015-05-18. [Online]. Available: https://webrtchacks.com/an-intro-to-webrtcs-natfirewall-problem/

[27] A. Johnston, J. Yoakum, and K. Singh, "Taking on WebRTC in an enterprise," *Communications Magazine, IEEE*, vol. 51, no. 4, pp. 48–54, 2013.

[28] C. J. A. N. Daniel Burnett, Adam Bergkvist. (2015, April) Media capture and streams. Last accessed: 2015-05-13. [Online]. Available: https://w3c.github.io/mediacapture-main/getusermedia.html

[29] S. Dutton. (2012, July) Getting started with WebRTC. Last accessed: 2015-05-13. [Online]. Available: http://www.html5rocks.com/en/tutorials/webrtc/basics/#toc-mediastream

[30] P. S. Inc. easyrtc documentation. Last accessed: 2015-04-24. [Online]. Available: http://easyrtc.com/docs/

A Appendix: getUserMedia and Libraries

This appendix will contain a detailed overview of how the getUserMedia object that WebRTC uses in order to get access to media devices is used. Also the three JavaScript libraries that are mentioned in the thesis is explained, the aim of the appendix is to give deeper knowledge on how they are used in practice and what their speciality is meant to be.

getUserMedia

It is necessary for WebRTC to use an object that handles media streams to be able to make video or audio calls. MediaStream is a Application Programming Interface (API) that represents synchronized streams of media [28] and describes the methods, constraints for the particular type of data and success, error callbacks when using the data asynchronously. Each MediaStream has an input which may be generated by **navigator.getUserMedia()**, and an output which might be passed to a html video element or a PeerConnection.

getUserMedia prompts the user for permission to use one video and/or audio input device such as a camera and/or microphone. If the user gives permission then a **successCallback** is invoked with the requested MediaStream object as parameter, if the user denies or if the media is unavailable, an **errorCallback** function is called with an PermissionDeniedError or a NotFoundError error object as parameter see Table 2.

Code-example A.1: Create getUserMedia example

```
navigator.getUserMedia(constraints, successCallback, errorCallback);
```

Code-example A.1 shows the syntax on how to use the getUserMedia object and get access to the wanted input devices. This will prompt the user who has to choose whether to accept or deny permission of the application to use the input devices, the prompt will be handled in a standard method for the specific browser used. On pages that use Hypertext Transfer Protocol Secure (HTTPS), permission has to be granted just once. MediaStream is intended to eventually be able to stream from any kind of source such as sensors or other inputs [29].

Table 2 shows a detailed view of each parameter that getUserMedia takes, if the parameter is required and a brief description of what it is. The constraints parameter is composed of text in JavaScript Object Notation (JSON) format, example shown in Code-example A.2.

Code-example A.2: Create constraints example

```
var constraints = { audio: true, video: { width: 1280, height: 720 } }
```

Code-example A.3 shows an example where getUserMedia is used, showcases usage and possible ways to structure code. The declaration of *navigator.getUserMedia* on line 1 shows how to handle different browsers implementations of getUserMedia.

Code-example A.3: getUserMedia Example

```
navigator.getUserMedia = navigator.getUserMedia ||
    navigator.webkitGetUserMedia ||
    navigator.mozGetUserMedia;

if (navigator.getUserMedia) {
    navigator.getUserMedia({ audio: true, video: { width: 1280, height: 720 }},
         function(stream) {
        var video = document.querySelector('video');
        video.src = window.URL.createObjectURL(stream);
        video.onloadedmetadata = function(e) {
            video.play();
        };
    },
    function(err) {
        console.log("The following error occured: " + err.name);
    });
} else {
    console.log("getUserMedia not supported");
}
```

Table 2 Parameters for getUserMedia

Parameter	Required	Description
constraints	Yes	A MediaStreamConstraints object specifying the types of media to request, along with any requirements for each type.
successCallback	Yes	The function to invoke with the resulting MediaStream object if the call succeeds.
errorCallback	Yes	The function to invoke with the resulting MediaStreamError if the call fails.

Peerjs

Explains how to use the library peerJs and how to create a WebRTC data-channel, multimedia call and how to send and receive data. Consists of examples and explanatory text. Some basic function calls to the library and explanations to them are also provided.

Code-example A.4 shows how the `peer` object may be created.

Code-example A.4: PeerJS: Create Peer; config with API key

```
var peer = new Peer( { key: 'myapikey' } );
```

To use peerJS, the library needs to be included in the header so that it loads directly. Either by downloading the peer.js file from the website or by linking to it in the HTML header, see Code-example A.5 for examples on both methods.

Code-example A.5: PeerJS: Loading peer.js

```
Method 1: <script src="http://cdn.peerjs.com/0.3/peer.js"></script>

Method 2: <script src="js/peer.js"></script>
```

Since it is a WebRTC library, a signaling server is needed. A developer can choose to setup an own peerJS signaling server with the signaling protocol of choice, or choose to use a free cloud-hosted peerJS signaling server that the developers of peerJS provides. This free server option currently has a maximum of 50 concurrent users.

Setting which STUN and TURN servers should be used by ICE is done by adding a config part to the configuration object passed on to `Peer` as shown in Code-example A.6.

Code-example A.6: PeerJS: Example on how to set STUN and TURN servers

```
var peer = new Peer({
  config: {'iceServers': [
    // Example servers
    { url: 'stun:stun.l.google.com:19302' },
    { url: 'turn:homeo@turn.bistri.com:80', credential: 'homeo' }
   }
});
```

DataChannel

To create a data connection between two peers, each peer needs to create a `Peer` object, as in Code-example A.4 or A.9. The `Peer` objects then communicates with the peerJS signaling server that will give each peer a unique ID, this is then used when connecting to other peers as Figure 7 illustrates. Code-examples A.7, A.8 shows how one peer connects to another and how the called peer receives the connection request.

Code-example A.7: PeerJS: Connect to a peer with PeerJS over RTCDataChannel

```
var conn = peer.connect('another-peers-id');
```

Code-example A.8: PeerJS: Receive Connection over RTCDataChannel

```
peer.on('connection', function(conn) { ... });
```

In Code-example A.4 the `Peer` object takes a configuration object in JSON just like we have seen other objects do in previous examples, in the configuration, information on which signaling server are going to be used needs to be provided. When using a private server the configuration needs to include ip and port to the server to be able to connect to it. On the other hand when using the cloud-hosted server the `Peer` object needs to take an API key which is mapped to you're very own cloud-hosted server. Examples can be seen in A.4, A.9.

Code-example A.9: PeerJS: Create Peer; config with host and port

```
var peer = new Peer( {
        host: 127.0.0.1,
        port: 1337
        } );
```

When the the WebRTC connection is open (after Code-example A.7, A.8) and peers want to send and receive data between each other the `conn` object has two functions, `conn.send(data)` (Code-example A.10) for sending and a EventHandler, `conn.on('data', function(data) { } )` Code-example A.11 to receive incoming data. These code examples checks so that the connection is active before trying to send or receive, with the code `conn.on('open')`

`{ ...` when sending and `peer.on('connection) {...` when receiving data which adds extra error-handling.

Code-example A.10: PeerJS Send data example

```
conn.on('open', function() {
    conn.send('hi!');
    });
```

Code-example A.11: PeerJS Receive data example

```
peer.on('connection', function(conn) {
    conn.on('data', function(data){
    // Will print 'hi!'
    console.log(data);
    });
});
```

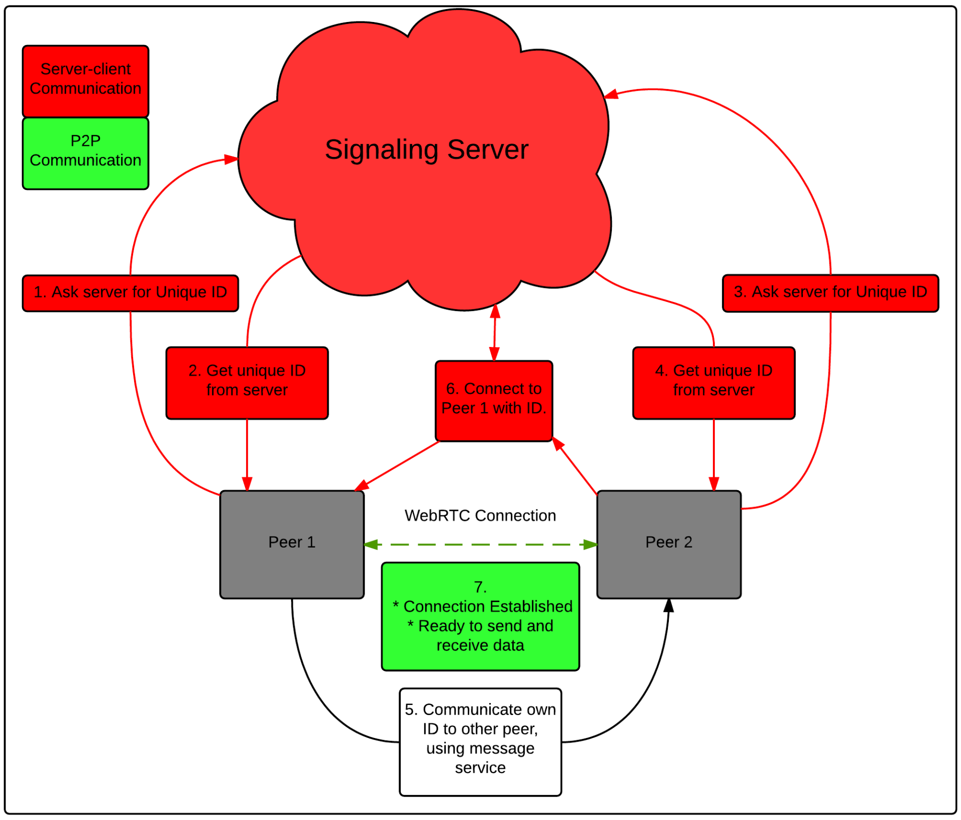
Server-client Communication
P2P Communication
Signaling Server
1. Ask server for Unique ID
2. Get unique ID from server
3. Ask server for Unique ID
4. Get unique ID from server
6. Connect to Peer 1 with ID.
Peer 1
Peer 2
WebRTC Connection
7. * Connection Established * Ready to send and receive data
5. Communicate own ID to other peer, using message service

Figure 7: *Step by step illustration of two peers setting up a WebRTC data communication between each other with the library PeerJS.*

Multimedia Calls

PeerJS tries to make media calls easier by hiding all the technicality of setting up a WebRTC connection. According to the creators of peerJS [15] a media call is created by using the *call*, *answer* functions that the library offers, Code-examples A.13, A.14. When answering, the `Peer` object has a EventHandler that listens for calls and when the event triggers, a callback function that takes the call as parameter is executed. This function then executes another function that the library provides that answers the call with the incoming stream object as parameter.

When a peer answers a call, (Code-example A.13). the mediaStream parameter is optional and if none is provided, a one-way call is established.

Code-example A.12: PeerJS: Get stream after answer

```
call.on('stream', function(stream) {
    // `stream` is the MediaStream of the remote peer.
    // Here you'd add it to an HTML video/canvas element.
});
```

`call.on('stream', function (mediaStream) {})` Code-example A.12 is an event that will be triggered when the callee have answered, the callback will include a mediaStream that is the audio/video of the remote peer, which is the mediaStream object and can be added to a HTML canvas or video element of the developers choice.

Code-example A.13: PeerJS: Simple Example Media Answer

```
peer.on('call', function(call) {
    // Answer the call, providing our mediaStream
    call.answer(mediaStream);
});
```

Code-example A.14: PeerJS Simple Example Media Call

```
// Call a peer, providing our mediaStream
var call = peer.call('dest-peer-id', mediaStream);
```

Function calls

Here follows a set of fundamental function calls to easyrtc and explanations (all calls are preceded by []peer.]):

- `call(id, stream [options])`: Calls the remote peer specified by id and returns a media connection. id is the ID of the peer, stream the callers media stream and options is metadata about the connection.
- `connect(id, [options])`: Connects to the remote peer specified by id, and returns a data connection.
- `disconnect()`: Close the connection to the peer.
- `reconnect()`: Attempts to reconnect to the peer with the old id.
- `destroy()`: Close the connection to the server and terminate all peer connections.
- `id`: Returns the ID of this peer.

EasyRTC

Explains how to use the library easyRTC and how to create a WebRTC dataChannel, multimedia call and how to send and receive data. Consists of examples and explanatory text. Some basic function calls to the library and explanations to them are also provided.

Code-example A.15: EasyRTC: Example of how to load easyRTC library on HTML page

```
<head>
    <script type="text/javascript" src="/easyrtc/easyrtc.js"></script>
</head>
```

EasyRTC uses unique id's called `easyrtcid` identify each peer and this id is supplied from the signaling server that the application uses [30]. The easyRTC signaling server that the developers of the library provides is able too have several different web applications of using the same signaling server, this is accomplished by having a `appName` for each application. This name is then sent from the peer side and the signaling server creates a session for each unique name, for every application that uses the server.

DataChannels

Using easyRTC to send and receive data is accomplished by using the `easyrtc` object and configuring it so that it only uses the data channel, and setting functions for receiving and sending the data. The data channel does not support sending a object of the dictionary data-type which is commonly used in JavaScript programming, dictionaries needs to be converted to strings and then sent by using JSON.stringify, unlike easyrtc.sendWS that supports sending dictionaries. Although it will use web-sockets instead of WebRTC.

Code-example A.16: EasyRTC Code snippets; Data Channel

```
easyrtc.enableDataChannels(true);

// disable video and audio channels, enabled by default
easyrtc.enableVideo(false);
easyrtc.enableAudio(false);
easyrtc.enableVideoReceive(false);
easyrtc.enableAudioReceive(false);

// Set a function to listen for incoming data
easyrtc.setPeerListener(addToConversation);

// Connect to signaling server
easyrtc.connect("application_name", loginSuccess, loginFailure);


function addToConversation(who, msgType, content) {
// do something with the received content in <content>.
}

// Call other peers with id = otherEasyrtcid
easyrtc.call(otherEasyrtcid, successCB, failureCB);

function sendData(msg) {
    if (easyrtc.getConnectStatus(otherEasyrtcid) === easyrtc.IS_CONNECTED) {
        easyrtc.sendDataP2P(otherEasyrtcid, msg, text);
    } else {
        easyrtc.showError("NOT-CONNECTED", "not connected to " + easyrtc.
            idToName(otherEasyrtcid) + " yet.");
    }
}
function loginSuccess(easyrtcid) {
    // Do something when login was succesfull.
}
function loginFailure(errorCode, message) {
    easyrtc.showError(errorCode, "failure to login");
}
```

Code-example A.16 shows code snippets on how to use easyRTC to create a peer-to-peer connection between two peers and send data between them. Instead of using sendDataP2P the function sendData could be used, it sends the data via the data channel if it is connected, otherwise it does fall back to using web-sockets for sending the data. All channels are enabled by default and this makes it the developers responsibility to disable the channels that is not intended to be used.

Multimedia Calls

For media calls easyRTC is used in a similar way as with the data channel, by using the `setVideoDims(width, height)` the size of the streams can be set or else default size of 720p dimensions will be used [30]. The `setRoomOccupantListener(function)` is an EventHandler that triggers the function it takes as parameter when a user enters the same room the user is in, the callback function is expected to take a room name and a datatype which have a key, value pair and where the key is a all users `easyrtcid` in the room that is going to be joined. The value is information specific to the user with the corresponding `easyrtcid` map where id's are easyrtcid's and the values are information specific to the user with that `easyrtcid`, a room in this context is the application name.

Code-example A.17: EasyRTC Media call example

```
var selfEasyrtcid = "";
function connect() {
    easyrtc.setVideoDims(640,480);

    // Set a function that triggers when a new user connects.
    easyrtc.setRoomOccupantListener(roomOccupantListener);

    // Set the name of the application, local stream, remote streams,
    // success and error functions
    easyrtc.easyApp("audioVideoExample", "selfVideo", ["callerVideo"],
        loginSuccess, loginFailure);
}

function performCall(otherEasyrtcid) {
    easyrtc.hangupAll();
    var successCB = function() {}; // Set to a success function
    var failureCB = function() {}; // set to a error handler
    easyrtc.call(otherEasyrtcid, successCB, failureCB);
}

function loginSuccess(easyrtcid) {
    selfEasyrtcid = easyrtcid;
    // Do other stuff when login is a success
}

// Error handler.
function loginFailure(errorCode, message) {
    easyrtc.showError(errorCode, message);
}
```

Function Calls

Here follows a set of fundamental function calls to easyrtc and explanations (all calls are preceeded by [easyrtc.]):

- `enableDebug(true/false)` : Set if debugging to the console should be on or off.

- `enableDataChannel(true/false)`: Set if RTCDataChannel should be used to send inter-peer messages. Only for messages that applications explicitly sends to other applications, not the WebRTC signaling messages.
- `enableVideo(true/false)`: Sets whether video is transmitted by the local user in any subsequent calls.
- `enableAudio(true/false)`: Sets whether audio is transmitted by the local user in any subsequent calls.
- `connect()`: Connects to a EasyRTC signaling server, must be connected before trying to connect to other users.
- `disconnect()`: Disconnects from the EasyRTC signaling server.
- `hangupAll()`: Disconnects peer from all current connections.
- `hangup(otherUser)`: Disconnect from a particular user.
- `setRoomOccupantListener(occupantListener)`: Sets a function to be called when the list of people logged in changes.
- `setVideoBandwidth(integer)`: Sets the maximum bandwidth used to send and receive a mediaStreams video track.
- `getConnectionCount()`: Returns the number of live peer connections the user has.

SimpleWebRTC

Explains how to use the library simpleWebRTC to create a WebRTC dataChannel, multimedia call and how to send and receive files. Consists of examples and explanatory text. Some basic function calls to the library and explanations to them are also provided.

Code-example A.18: SimpleWebRTC: Include simpleWebRTC library

```
<head>
Method 1: <script src="//simplewebrtc.com/latest.js"></script>

Method 2: <script src="/js/simplewebrtc.js"></script>
</head>
```

In order to be able to use simpleWebRTC, an `SimpleWebRTC` object is first required to be created, Code-example A.19 illustrates how this can be done. The `SimpleWebRTC` object takes a a configuration file which is in JSON format, and in the example the localVideoEl, and the remoteVideosEl are set to their respective HTML elements cause it is a multimedia connection.

- `[autoRequestMedia: true]`: Makes the application immediately ask for camera acces.
- `[url: 'URL']`: URL to the signaling server that is going to be used by the application.

Code-example A.19: SimpleWebRTC: Create simpleWebRTC object example

```
var webrtc = new SimpleWebRTC({
    // the id/element dom element that will hold "our" video
    localVideoEl: 'localVideo',
    // the id/element dom element that will hold remote videos
    remoteVideosEl: 'remotesVideos',
    // immediately ask for camera access
    autoRequestMedia: true
    url: 'https://example-signaling-server.com/'
});
```

DataChannel

When establishing a WebRTC data channel by using simpleWebRTC, instead of specifying HTML elements for local and remote video, since they will not be needed. They are not set to any element. Since audio or video is not needed, `autoRequestMedia` is set to false cause when creating a data channel the user does not need to agree to a camera or microphone to be used. Code-example A.20 shows an example of this and also how to bypass so that media does not need to be negotiated.

Code-example A.20: SimpleWebRTC: Create simpleWebRTC object; data channel

```
var webrtc = new SimpleWebRTC({
    // we don't do video
    localVideoEl: '',
    remoteVideosEl: '',
    // dont ask for camera access
    autoRequestMedia: false

    // dont negotiate media
    receiveMedia: {
        mandatory: {
            OfferToReceiveAudio: false,
            OfferToReceiveVideo: false
        }
    }
});
```

Since there is no need to wait for any media to be ready when initializing a WebRTC data channel, a room can be joined directly as in Code-example A.21. In the example a room is joined and a connection is established between every peer that is in the room and an EventHandler, `webrtc.on('createdPeer, function(peer) {..} )` is triggered for every peer connection that is created shown in Code-example A.21.

Code-example A.21: "simpleWebRTC: joinRoom example"

```
// join without waiting for media
webrtc.joinRoom('your awesome room name');
```

This will join the room and create a connection with every peer that joins. So let's wait for a peer to join Code-example A.22:

Code-example A.22: "simpleWebRTC: Peer joins room"

```
// called when a peer is created
webrtc.on('createdPeer', function (peer) {
    console.log('createdPeer', peer);
}
```

Send file

Sending a file with simpleWebRTC seems easy and straightforward, after allocating something to send, either via a user choosing a file to send in some manner. All that needs to be done is call the `peer.sendFile(fileToSend)`, Code-example A.23 to send the file to that specific peer, the return value of the `sendFile` call is an object containining three events:

- *process:* Process can be attached to a progress bar to show the progress.
- *sentFile:* Indicates that the sender considers the transfer to be complete, due to buffering it is not safe to close the connection yet.
- *complete:* The transfer is complete and it is safe to close the connection.

Code-example A.23: ”simpleWebRTC: Send file to peer”

```
    var sender = peer.sendFile(file);
});
```

Receive file

When receiving a file, an EventHandler that triggers on incoming filetransfer , the event is called with metadata bout the file, such as the name and size and a receiver object as seen in Code-example A.24. The receive object emits two events:

- *progress:* In the same manner as when sending a file this can be attached to a HTML <progress> element.
- *receivedFile:* Called when the transfer is complete. parameters for the function is the file and its metadata.

Code-example A.24: ”simpleWebRTC: Receive file from peer”

```
// receiving an incoming filetransfer
peer.on('fileTransfer', function (metadata, receiver) {
console.log('incoming filetransfer', metadata.name, metadata);
receiver.on('progress', function (bytesReceived) {
console.log('receive progress', bytesReceived, 'out of', metadata.size);
});

// get notified when file is done
receiver.on('receivedFile', function (file, metadata) {
console.log('received file', metadata.name, metadata.size);

// close the channel
receiver.channel.close();
});
filelist.appendChild(item);
});
```

Multimedia Calls

Media calls with simpleWebRTC is streamlined so that with only a few steps multimedia communication via WebRTC can be established. Firstly a `SimpleWebRTC` object needs to be created, Code-example A.25 shows an example of how to set up a video call. The `SimpleWebRTC` object has a localVideoEl and a remoteVideoEl for the HTML elements that is going to be used to show the video. Camera access is requested immediately and how to specify the signaling server to be used is shown in the example.

Code-example A.25: ”simpleWebRTC: Media Call; create simpleWebRTC object example”

```
var webrtc = new SimpleWebRTC({
    // the id/element dom element that will hold "our" video
    localVideoEl: 'localVideo',
    // the id/element dom element that will hold remote videos
    remoteVideosEl: 'remotesVideos',
    // immediately ask for camera access
    autoRequestMedia: true
    url: 'https://example-signaling-server.com/'
});
```

After the `SimpleWebRTC` object is initialized its time to join a room, simpleWebRTC has a eventHandler exemplified in Code-example A.26, that gets triggered when the user has agreed to share the media devices, then `webrtc.joinRoom('room name')` is used to join a room.

Code-example A.26: ”simpleWebRTC: Join a room when ready example”

```
// we have to wait until it's ready
webrtc.on('readyToCall', function () {
    // you can name it anything
    webrtc.joinRoom('your awesome room name');
});
```

Function Calls

Here follows a set of fundamental function calls to simpleWebRTC and explanations (all calls are preeceded by [simplewebrtc.]):

- `unmute()`: Unmutes sound.
- `mute()`: Mutes sound.
- `pauseVideo()`: Pauses the video streaming.
- `resumeVideo()`: Resumes playback.