Fachhochschule
Münster University of
Applied Sciences

Bachelor Thesis

by

Lennart Grahl <lennart.grahl@gmail.com>

SaltyRTC

Seriously Secure WebRTC

Münster University of Applied Sciences

Electrical Engineering and Computer Science Faculty

19 October 2015

Eidesstattliche Erklärung

Ich versichere, dass ich diese schriftliche Arbeit selbständig angefertigt, alle Hilfen und Hilfsmittel angegeben und alle wörtlich oder im Sinne von Veröffentlichungen oder anderen Quellen, insbesondere dem Internet entnommenen Inhalte, kenntlich gemacht habe.

_____

Abstract

WebRTC is an API definition by the World Wide Web Consortium (W3C) which obtained a high popularity among browser-to-browser applications. One of the core reasons for that popularity is the simplicity it provides to set up a peer-to-peer connection. Although it is often mainly used for audio and video communication, there is also a *Data Channel* that allows arbitrary bidirectional data transfers between two peers. *Datagram Transport Layer Security* (DTLS), which is based on *Transport Layer Security* (TLS), is being used as WebRTC's security layer.

In the recent past, plenty of vulnerabilities in TLS have been revealed and it is reasonable to assume that there are more to come. In addition, WebRTC API users have no control over the key pair generation, nor how the public keys are being exchanged between the peers. However, even more problematic is that WebRTC requires an implementation of a *Signalling Channel* to exchange metadata that is required to set up a peer-to-peer connection. This metadata already contains security-relevant information that no third party should be able to read or modify, unless it is absolutely necessary for the use case.

This work presents a solution that uses the *Networking and Cryptography library* (NaCl) to provide another security layer for WebRTC Data Channels and a secure implementation of the *Signalling Channel*. The developed software collection is called *SaltyRTC* and will be released on GitHub soon.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Structure 1

2 Background **2**

2.1 WebSocket 2

2.2 WebRTC 4

2.3 NaCl 13

3 Security Evaluation of WebRTC **15**

3.1 Signalling 16

3.2 Peer Connection Authentication 16

3.3 Replay Attacks 16

3.4 DTLS Encryption 17

3.5 Conclusion 17

4 SaltyRTC - Secure WebRTC based on NaCl **17**

4.1 Terminology 18

4.2 Architecture 20

4.3 Exchanging the Authentication Token and the Public Key 22

4.4 Signalling Channel 22

4.5 Data Channel 31

4.6 Peer-to-Peer Connection Build-Up 32

4.7 Security Analysis 32

4.8 Possible Improvements 34

5 Threema Web Client Prototype **35**

5.1 Introduction 36

5.2 Analysis 36

5.3 Solution 36

5.4 Prototype 37
5.5 Conclusion 40

6 References **41**

List of Figures

1 WebRTC Architecture [5] . . . 5
2 WebRTC Protocol Stack . . . 7
3 WebRTC Data Paths . . . 11
4 Simple Relay Signalling Channel . . . 12
5 NaCl Public-Key Authenticated Encryption Example . . . 14
6 SaltyRTC Nonce Structure . . . 19
7 SaltyRTC Architecture . . . 21
8 Signalling Channel Packet Structure . . . 23
9 SaltyRTC Signalling Channel Server Authentication . . . 29
10 SaltyRTC Signalling Channel Peer Authentication . . . 30
11 Data Channel Packet Structure . . . 31
12 Threema Web Client Communication . . . 38
13 Threema Web Client Prototype . . . 39

List of Tables

1 Comparison of Transport Layer Protocols [19, chap. 18 table–18–1] . . . 9
2 Possible Data Channel Configurations [19, chap. 18 table–18–3] . . . 13
3 Receiver Types and Values . . . 23

Glossary

Application Programming Interface (API)

Defines inputs, outputs and underlying types. The interface is independent of the implementation.

Data Channel

Channel for arbitrary data transfer in WebRTC.

Diffie-Hellman Key Exchange

A method of securely exchanging cryptographic keys over a public channel.

Datagram Transport Layer Security (DTLS)

A collection of cryptographic protocols for datagram protocols. Based on TLS.

JavaScript Object Notation (JSON)

An open format that uses human-readable text to encode data objects consisting of key-value pairs.

Long Polling

A technique for HTTP requests where the response is held back until data is available.

Message Authentication Code (MAC)

A short piece of information used to provide integrity and authenticity assurances for a message.

Networking and Cryptography Library (NaCl, pronounced *salt*)

A software library for network communication, encryption, decryption, signatures and various utility functions.

Nonce

An arbitrary number that may only be used once.

Peer-to-Peer Connection

A direct connection between two communication partners (peers).

Forward Secrecy

A property of protocols which ensures that a temporary session key is being established over a set of long-term keys.

RSA

An asymmetric cryptosystem based on the factoring problem, designed by Ron Rivest, Adi Shamir and Leonard Adleman.

Real-time Transport Protocol (RTP)

A protocol for efficient audio and video transmission in real-time.

Signalling Channel

A channel for exchanging the session description and network reachability of a peer.

Secure Real-time Transport Protocol (SRTP)

An AES encrypted profile for the Real-time Transport Protocol.

Session Traversal Utilities for NAT (STUN)

A network protocol to allow an end host to discover its public IP address if it is located behind a NAT.

Traversal Using Relays around NAT (TURN)

A protocol that assists in traversal of network address translators (NAT) or firewalls. TURN servers are able to relay arbitrary data in case of a strict NAT.

Transport Layer Security (TLS)

A collection of cryptographic protocols designed to provide communications security over a computer network.

Uniform Resource Identifier (URI)

A string of characters used to identify the name of a resource.

World Wide Web Consortium (W3C)

The main international standards organisation for the World Wide Web.

Web Real-Time Communication (WebRTC)

An API definition of the W3C that primarily supports browser-to-browser applications for media and file sharing.

WebSocket

A protocol providing full-duplex communication channels over a single TCP connection.

XMLHttpRequest

An API interface in the browser to send HTTP requests to a web server and retrieve the server's response.

1 Introduction

1.1 Motivation

In the last decade, the number of devices per person have increased dramatically. These devices all gather and store data that users want to access on all of their devices. Consequently, plenty of applications exist that require a secure data communication layer between two endpoints. Although it is not required, most of the time the data is relayed over a server. But for confidential data, the shortest route from one endpoint to another is preferable. Thus, relaying should be avoided. The main reason behind this common practice is the possibility to store data on an always-online server, so devices that synchronise their data do not have to be online at the same time.

But another reason is Network Address Translation (NAT) which prevents most incoming connections that have not been established from within the network (e.g. behind the NAT). There are various methods to mitigate this problem. But on their own they never achieved a solution to all problems that NAT exposes. Interactive Connectivity Establishment (ICE) [1] is a technique that combines these various methods to overcome the described problems. However, due to ICE's complexity, it has not been used in many protocols, yet.

WebRTC is an API definition still in development by the World Wide Web Consortium (W3C) which allows the browser to establish a peer-to-peer connection, by the help of ICE, without the need for a browser plugin. Furthermore, audio, video and arbitrary data can be exchanged easily over this peer-to-peer connection and WebRTC does all the hard work for us. Although WebRTC is especially designed for the use in a browser, it is also possible to write Android and iOS applications that use WebRTC.

What was very difficult in the past is now achievable for every application developer. WebRTC opens the doors for peer-to-peer applications over the shortest route that can be used on all major operating systems and devices.

This bachelor thesis will explore the technologies that are required and highlight potential security issues of WebRTC. We have developed a software collection, called *SaltyRTC*, which simplifies the usage and tries to overcome the security issues of WebRTC by using the Networking and Cryptography library (NaCl).

1.2 Structure

Before we can set up a WebRTC-based peer-to-peer connection, session information has to be exchanged. SaltyRTC exchanges these information by using WebSocket, therefore we will go through the basics of WebSocket followed by the basics of WebRTC. Because SaltyRTC uses NaCl, we will take a short look at NaCl as well. Afterwards, the software collection SaltyRTC

will be introduced. In this part, the protocol design of SaltyRTC's signalling mechanism will be explained followed by the data channel protocol.

2 Background

2.1 WebSocket

In this section, we will go through the most important features and properties of WebSocket that are required to understand the design choices for the signalling channel implementation of SaltyRTC.

2.1.1 Introduction

WebSocket is a protocol standardised by the IETF as RFC 6455 [2] in 2011 which enables bidirectional communication from a client to a remote host. The corresponding WebSocket API has been drafted by the W3C [3] as part of the HTML5 specification. The core reason for developing WebSocket was to provide a protocol that natively supports bidirectional data transfers in the browser while not relying on multiple *XMLHttpRequest*s or *long polling*. WebSocket uses a single TCP connection for traffic in both directions. This circumvents a variety of problems that existed formerly with HTTP polling, e.g. the usage of a TCP connection for each incoming message or the overhead of the HTTP header for each message. Similar to HTTPS (HTTP over TLS), *TLS* (Transport Layer Security) can be used to encrypt the communication on the WebSocket.

2.1.2 Handshake

The handshake of WebSocket is rather unusual because the HTTP protocol is being used at the beginning. Afterwards, the protocol is switched over to the binary protocol of WebSocket. However, this ensures that the ports 80 and 443 can be used which are sometimes the only open destination ports for clients.

To establish a WebSocket connection, the client sends a HTTP *GET* request to the server along with a Path, the *Connection* header set to the value *Upgrade* and various other custom WebSocket headers. These are used for negotiation purposes and include information such as the client version or the supported subprotocols. The header *Sec-WebSocket-Key* contains a generated key which acts as a challenge to the server.

```
GET /0187700f92be19782443836d8b21c564b4988bc6b075ee03fabb6a21453b517d HTTP/1.1
Host: zwuenf.net:8765
Connection: Upgrade
Upgrade: websocket
Origin: http://zwuenf.net
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: Ej4YgAECgmZjG4xWdxIV8w==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
...
```

The server sends a HTTP response back to the client. In case the server accepts the connection requests, he will respond with status code *101 Switching Protocols* and the header *Upgrade* set to *websocket*. Again, custom WebSocket headers may be sent in the response. If the client sent information about which subprotocols he understands, the server must select one of these subprotocols or reject the connection request. In the header *Sec-WebSocket-Accept* the server proves that he supports the requested protocol version. The challenge of the client is concatenated with a unique GUID defined in the WebSocket standard and hashed afterwards. This mechanism prevents proxies from blindly accepting WebSocket upgrade requests when they do not understand the protocol.

```
HTTP/1.1 101 Switching Protocols
Server: Python/3.4 websockets/2.4
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Accept: 971+drIT4NV5JEzepMIeULd/46k=
```

2.1.3 Path

The path value is a simple string that is specified in the client's WebSocket request. Although the purpose of the path is not specified, the server could use the path to distinguish clients and let those clients that use the same path communicate with each other.

2.1.4 Payload Types

The WebSocket protocol allows to send text (UTF-8 encoded) and binary data. In the browser, the API automatically converts the message to the correct type, e.g. a *DOMString* object or a *Blob* object. Instead of the *Blob* object type, the developer may request that binary should be converted into *ArrayBuffer* objects. On the one hand, *ArrayBuffer* object allows modifying and slicing the data into smaller chunks. On the other hand, the *Blob* object is immutable but provides better performance.

2.1.5 Keep-Alive

To do connection liveness checks on an established WebSocket connection, client and server may send the a *ping* message to the other peer. This message contains a payload that the receiver has to repeat. The receiver of the *ping* then has to answer with a *pong* message.

The API in the browser does not provide any control over this mechanism. Therefore, it is not possible to manually send *ping* messages to the server. Other implementations that can be used outside the browser usually have an API that allows sending these messages manually.

2.2 WebRTC

WebRTC is an API definition drafted by the W3C [4] that allows real-time communication between browser and mobile applications. The underlying protocols used are standardised by the IETF. Modern browsers support this technology natively without the requirement of a plugin. It can be used to exchange audio, video and arbitrary data directly from peer to peer without usage of an external server. All data is being encrypted by either SRTP (for real-time data) or DTLS (arbitrary data over SCTP). In addition, all required audio and video engines with all their complexities are integrated into WebRTC, so no external software needs to be installed. API abstractions unify these engines and make sure that they can be used easily.

The peer connection between two peers is being established on multiple routes and the *best* (short and stable) route is being used for communication. The route may change on demand in case the currently used one is becoming unstable or a better route has been found. WebRTC is a technology, not a solution. Thus, exchanging session information and implementing authentication is up to the developer.

To understand the design choices that have been made for *SaltyRTC*, we first have to go through some of the core elements of WebRTC. Nevertheless, it is important to keep in mind that WebRTC is still in development and all characteristics described here are subjects to change.

2.2.1 Architecture

In the following, we will describe the various sections shown in figure 1.

2.2.1.1 Web API The *Web API* is an API to be used by web developers. This is the most abstract interface WebRTC offers and the only interface that can be accessed in the browser. Because the API is currently still in development, browsers that support WebRTC have different vendor specific names for functions, etc. However, there is an API adapter script maintained by Google that unifies the differences between browsers. [6]

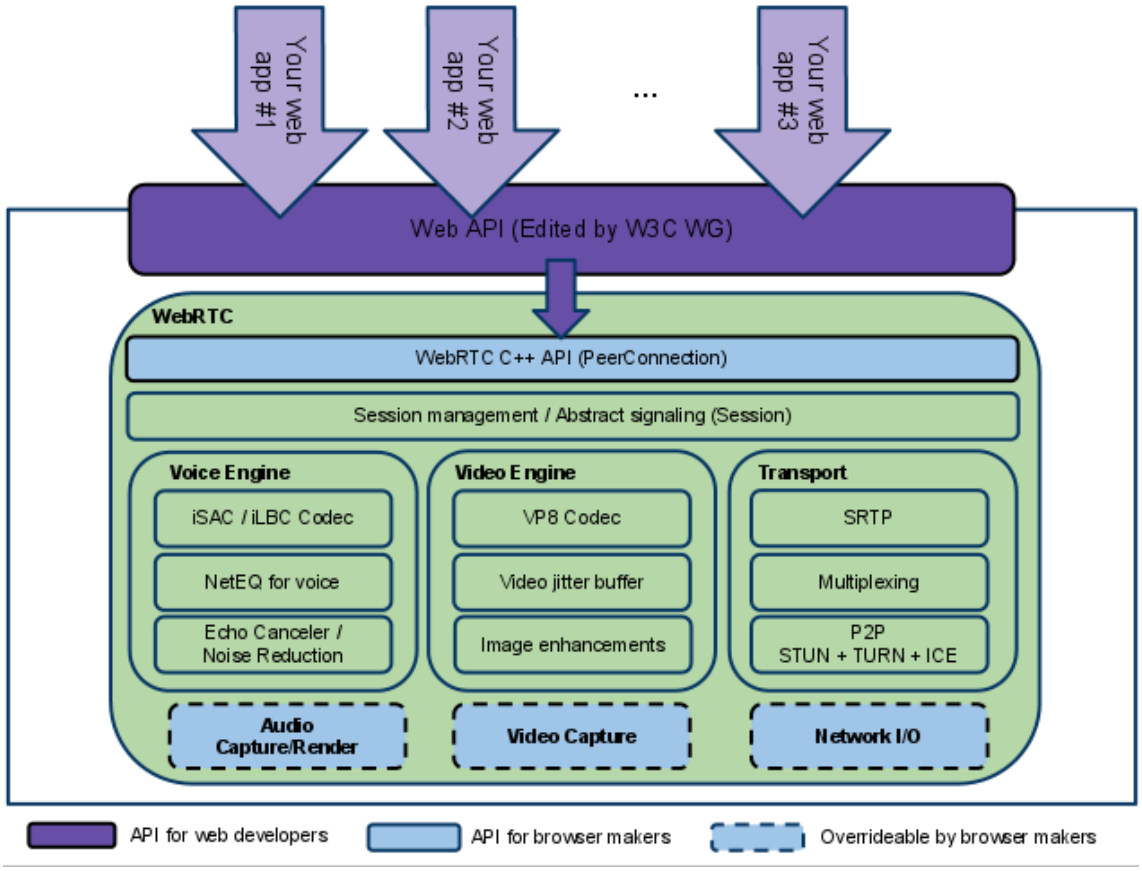

Figure 1: WebRTC Architecture [5]

WebRTC is in development since 2011. Currently, recent versions of Chrome, Opera, Bowser and Firefox support the most vital features of WebRTC. Edge lacks support for Data Channels while the Internet Explorer and Safari do not support WebRTC, at all. Despite that, there are third party plugins which add WebRTC support to Safari and Internet Explorer. [7]

2.2.1.2 WebRTC Native C++ API The *WebRTC Native C++ API* is an API layer for browser developers, so they can implement the API definition of the W3C. It is also being used by the less known Android and iOS libraries which basically wrap Java/Objective C around the C++ API.

2.2.1.3 Session Management This is an abstract session layer which leaves the protocol implementation to the application developer. It provides access to various management functionalities which are required to set up a WebRTC session.

2.2.1.4 Voice Engine The voice engine unifies various audio codecs and algorithms to handle and potentially counteract network jitter and high latency while maintaining a high voice quality. It also tries to filter out the acoustic echo from the speaker into the active microphone and several types of background noise.

2.2.1.5 Video Engine The video engine includes the video codecs and, much the same as with the audio engine, provides algorithms to conceal the effects of network jitter and packet loss on video quality. Additionally, it includes image enhancements which removes video noise from captured images.

2.2.1.6 Transport To establish a connection between two peers, the section *Transport* in figure 1 includes the protocols ICE, STUN and TURN. These are vital components to establish a direct communication channel between two peers and will be described in the Signalling chapter. Moreover, multiplexing and the RTP (Real Time Protocol) network stack are being handled in the *Transport* section.

2.2.2 Protocol Stack

Understanding how WebRTC establishes a peer connection and how the protocols are being intertwined are the core components that need to be comprehended. Therefore, we will go through and explain all important protocols and techniques used, including an evaluation of their security mechanisms where meaningful.

The protocols shown in figure 2 will be explained from bottom to top. The Network (IP) layer will be skipped.

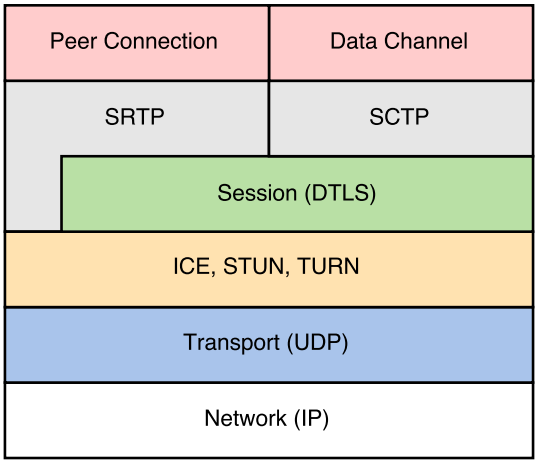

Figure 2: WebRTC Protocol Stack

2.2.2.1 UDP The User Datagram Protocol (UDP) [8] is a transport layer protocol on top of the IP layer and one of the core members of the IP suite. UDP follows a connectionless model and is unreliable. Accordingly, there is no guarantee of delivery, order or protection against duplicate packets. In case of WebRTC, UDP is preferable over other transport layer protocols for several reasons:

1. Traversing most NATs is possible with UDP by a technique called *UDP Hole Punching* which is being used by the ICE protocol.
2. On the one hand, UDP is neither reliable nor does it ensure that packets being received are in order. On the other hand, it guarantees a low latency which is desirable for real-time audio and video communication where small errors do not matter much but a high latency due to packet retransmission is intolerable.
3. All common consumer level routers handle UDP packets. This is not the case for less common protocols like SCTP.

2.2.2.2 STUN The Session Traversal Utilities for NAT (STUN) [9] is a protocol that allows an end host to discover its public IP address in case it is located behind a NAT. It is designed to be used as a tool by other protocols such as ICE. STUN requires at least one external server to discover a peer's public IP address and can use multiple protocols, including UDP, TCP and TLS over TCP.

2.2.2.3 TURN Traversal Using Relays around NAT (TURN) [10] is a relay extension to STUN. Some types of NAT make it impossible to establish a direct communication between two peers.

In this case, the affected peer needs an intermediate node which acts as a communication relay. The TURN specification defines a protocol that allows the peer to control the relay operation and to exchange arbitrary data over the relay node. Communication with multiple peers is possible using a single relay address. TURN can be used over either TCP or UDP.

For IPv6 support, a TURN Extension for IPv6 [11] has been defined which includes IPv4-to-IPv6, IPv6-to-IPv6 and IPv6-to-IPv4 relaying.

2.2.2.4 SDP The Session Description Protocol (SDP) [12] is intended for describing multimedia communication sessions. Its main purpose is to announce and invite to a media communication session and negotiate its parameters. The various parameters may be media types, formats, etc. In case of the offer/answer model [13], it is also being used for exchanging network reachability information.

2.2.2.5 ICE The Interactive Connectivity Establishment (ICE) [1] is a protocol for NAT traversal. ICE utilises STUN and its extension TURN to establish a peer-to-peer connection using the offer/answer model [13] which is based on the Session Description Protocol (SDP). It can be used by any protocol that supports the mentioned model. Moreover, it supports address selection for multi-homed and dual-stack hosts because ICE exchanges multiple IP addresses and ports.

Both peers supply several IP addresses and ports in SDP offers and answers. Offers and answers need to be exchanged via a signalling protocol which is not specified. The IP addresses and ports are tested for connectivity by STUN. TURN is being used in case a direct connection is not achievable.

2.2.2.6 DTLS The Datagram Transport Layer Security (DTLS) protocol [14] is a derivation of the TLS protocol which provides the same security services for unreliable protocols such as UDP. A minimal amount of changes has been made to TLS to support unreliable protocols.

Because the Secure Real-time Transport Protocol has no defined mechanism to exchange AES keys, the Datagram Transport Layer Security Extension DTLS-SRTP [15] makes use of the DTLS handshake to establish keying material, algorithms and parameters for SRTP.

2.2.2.7 RTP The Real-time Transport Protocol (RTP) is a protocol for efficient audio and video transmission in real-time. RTP is accompanied by the RTP Control Protocol (RTCP).

2.2.2.8 RTCP The RTP Control Protocol (RTCP) [16] monitors and controls the media streams that RTP transmits.

2.2.2.9 SRTP The Secure Real-time Transport Protocol (SRTP) [17] is an AES encrypted profile for RTP which provides confidentiality, message authentication and replay protection to RTP streams. It is designed to have a low overhead and a small footprint which makes it suitable for transmitting encrypted audio and video data in real-time. However, SRTP does not define a mechanism on how to exchange the AES keys from one peer to another.

2.2.2.10 SRTCP The Secure RTCP (SRTCP) [17] protocol is a sister protocol of SRTP which provides the same security features as SRTP to the unencrypted RTCP protocol.

2.2.2.11 SCTP The Stream Control Transmission Protocol (SCTP) [18] is a transport-layer protocol which is connectionless and message-oriented like UDP but can be reliable like TCP. Unlike UDP, it provides a flow and congestion control. SCTP can be configured to require packet ordering or support unordered messages. Moreover, it supports multi-homing and multiple unidirectional data streams. Because of its flexibility, SCTP is an ideal candidate for WebRTC whose Data Channel API allows to configure the underlying SCTP stack for reliability and ordered packets or unreliable and unordered data transmission. Additionally, the availability of multiplexing makes it possible to have multiple independent data channels which is a requirement of the WebRTC Data Channel API.

The reason why SCTP was an obvious choice over UDP or TCP can be seen in table 1.

	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

Table 1: Comparison of Transport Layer Protocols [19, chap. 18 table–18–1]

SCTP is the most configurable protocol and provides both flow and congestion control.

2.2.2.12 Interplay of the Protocols UDP is the main transport protocol used for all communication of WebRTC. ICE, by utilising STUN and TURN, has the task to set up a peer-to-peer connection. The necessary information to set up this peer-to-peer connection are being described using the offer/answer model with SDP and transferred on the signalling channel.

DTLS, with the DTLS-SRTP extension, is being used for both SRTP key establishment and as an encryption layer for SCTP packets. After the keying procedure, the DTLS session is no longer required for data transferred over SRTP until re-keying is necessary. However, this does not concern the tunnelled SCTP packets which still require DTLS.

SRTP, SRTCP and SCTP are the application protocols. Audio and video data is transferred with SRTP while arbitrary data is being transferred via data channels over SCTP. Usually, SRTP and SRTCP have separate ports which would be difficult for peers behind NATs. Therefore, WebRTC uses a multiplexing extension to allow multiple streams and their control channels on the same destination port.

2.2.3 RTCPeerConnection Object

The *RTCPeerConnection* object is the API entry point for all WebRTC communication. Creating a *RTCPeerConnection* object requires the configuration of *STUN* or *TURN* server Uniform Resource Identifiers (URIs). Additionally, constraints for media and various other options can be supplied. An RSA key pair will be automatically generated for each *RTCPeerConnection* instance.

To establish a peer-to-peer connection, first of all, an *offer* SDP message has to be created. The *offer* message contains, among other things, types of media to be exchanged, the key fingerprint of the generated RSA key pair and an ICE password. This *offer* message needs to be sent to the other peer over a so called Signalling Channel.

When the other peer receives the *offer* message, an *answer* SDP message needs to be created from the information of the *offer* message. This *answer* message will then be sent back to the other peer over the *Signalling Channel*.

Meanwhile, both peers' ICE Agents gather so called *ICE candidates*, which contain network reachability information of a peer. Acquiring the network reachability information requires a *STUN* or *TURN* server. However, there are plenty of publicly accessible *STUN* and *TURN* servers with no authentication requirement. Again, these candidates are being exchanged over the *Signalling Channel*. The gathering process happens in the background because querying the external IP addresses may take a bit of time. Each available candidate triggers an event. Therefore, ICE candidates, that can be retrieved quickly, can be sent to the other peer which may speed up initiating the peer connection. This technique is called *Trickle ICE* and is currently proposed as an extension to the ICE protocol. [20]

As soon as the ICE agent has received sufficient *ICE candidates*, all requirements for a peer-to-peer connection are satisfied and the peer-to-peer connection will be initiated, in some cases with the help of a relay server. If no sufficient *ICE candidates* have been found, the peer connection fails and the whole process needs to be restarted from the beginning.

Now, depending on the configuration, media streams and Data Channels can be created and added to the peer connection. The other peer receives these objects by incoming events. Multiple peer connections can coexist and can be used at the same time to allow telephone conferences, etc.

The various paths, on which data may be transmitted, can be seen in figure 3.

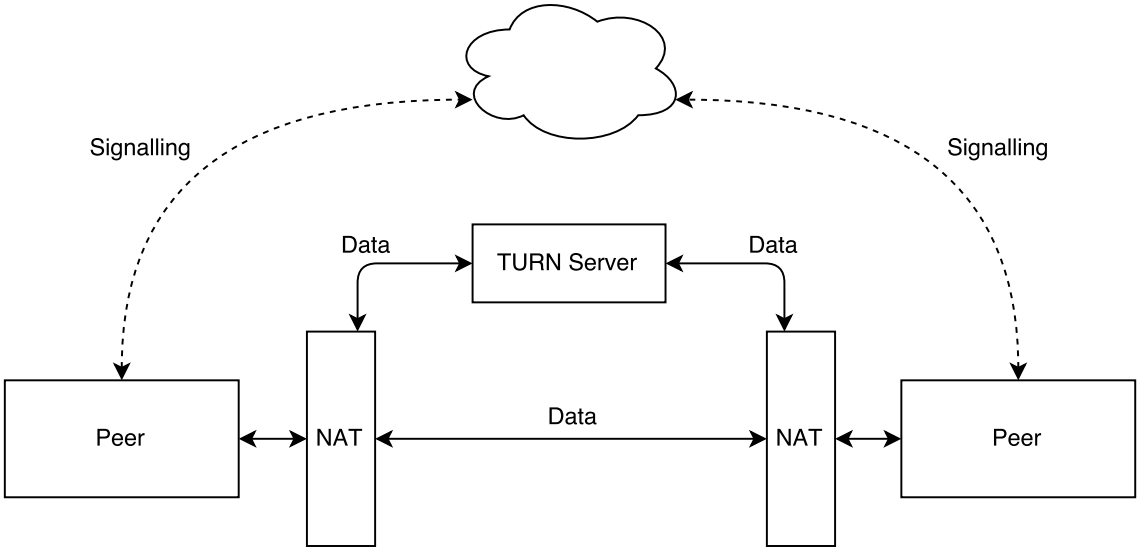

Figure 3: WebRTC Data Paths

2.2.4 ICE Agent

An ICE agent is responsible for gathering and managing *ICE Candidates* (which are IP, port tuples). These candidates are being queried from *STUN* or *TURN* servers and contain network reachability information. The agent prioritises *ICE candidates*, sets up and monitors connections between the peers and keeps the connection alive. Keep-alive messages are especially vital for UDP connections that have been established by *UDP Hole Punching*. At any time, the agent may decide to try other or even re-gather *ICE Candidates* in case the current connection becomes unstable or does not fit the current requirements for throughput or latency.

2.2.5 Signalling Channel

As mentioned before, setting up a WebRTC peer-to-peer session requires an implementation of the *Signalling Channel* in order to exchange *offer*/*answer* and network reachability information. Apart from the requirement that both peers can create and exchange *offer* and *answer*

SDP messages, there is no further constraint or specification on how to exchange this information. Consequently, ensuring confidentiality, integrity and authentication of this channel is up to the developer.

Having said this, the actual techniques for a signalling channel implementation, which works in the browser, are limited. Basically, it boils down to three different techniques:

1. XMLHttpRequest: Unidirectional protocol. Sending and receiving messages as part of an HTTP request or response.
2. WebSocket: Bidirectional communication protocol that uses an HTTP friendly handshake.
3. Third party plugins that allow access to other protocols or even sockets.

The simplest implementation of such a channel would require that both peers connect to a server. Messages that need to be exchanged on the signalling channel will simply be relayed by the server from one peer to another. A visual representation of the communication can be seen in figure 4:

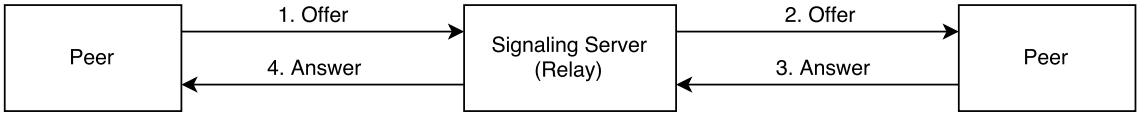

Figure 4: Simple Relay Signalling Channel

1. Send an *offer* to the signalling server.
2. Forward the *offer* to the other peer.
3. Send an *answer* to the signalling server.
4. Forward the *answer* back to the peer who sent the *offer*.

The same procedure would take place for exchanging ICE candidates until the gathering is complete. However, the channel should remain open in case the ICE agent decides to re-gather candidates. An alternative approach would be to switch signalling communication over to a data channel as soon as the peer connection is established.

There are countless of options for signalling channel implementations. For example, telephone providers could implement a gateway which translates Session Initiation Protocol offers to WebRTC offers and vice versa.

2.2.6 Data Channel

Once the peer connection is established, the peers can create multiple data channels to exchange arbitrary data. Every data channel has its own identifier and can be configured with custom delivery and reliability semantics. Because all data channels are multiplexed over the same SCTP association, head-of-line blocking can be avoided between the different streams and simultaneous delivery is guaranteed.

Data channels can be configured to be ordered or unordered and reliable or unreliable. These attributes can be combined freely and supply a flexibility UDP and TCP cannot provide. Furthermore, for unreliable channels, either the maximum packet life time or the maximum amount of retransmits can be specified. The result is a partially reliable delivery channel with either retransmit or timeout. All in all, there are six different configurations that are shown in table 2.

Configuration	**Ordered**	**Reliable**
Ordered & Reliable	yes	yes
Unordered & Reliable	no	yes
Ordered & Partially Reliable with Retransmission Counter	yes	partial
Unordered & Partially Reliable with Retransmission Counter	no	partial
Ordered & Partially Reliable with Timeout	yes	partial
Unordered & Partially Reliable with Timeout	no	partial

Table 2: Possible Data Channel Configurations [19, chap. 18 table–18–3]

The first configuration, *Ordered & Reliable*, is equivalent to TCP. Although the equivalent to UDP is not so obvious, it can be found in the fourth configuration, *Unordered & Partially Reliable with Retransmission Counter*, with the maximum amount of retransmits set to zero.

2.3 NaCl

SaltyRTC uses the so called *box* model of the Networking and Cryptography Library (NaCl) [21] to encrypt and authenticate messages. NaCl (pronounced *salt*) has been designed to be easy-to-use and to provide high-speed and high-security. Its fundamental operation is the public-key authenticated encryption which ensures confidentiality, integrity and authentication of a message between sender and receiver.

2.3.1 Box

Other cryptographic libraries that provide public-key authenticated encryption typically require several steps to prepare an encrypted and authenticated message. The *box* model of NaCl expresses a high-level functionality that does everything in one step. It converts a series of bytes into a *boxed* packet that is protected against espionage and unnoticeable modification.

2.3.2 Public-Key Authenticated Encryption

The used cryptographic tools are a combination of the Curve25519 Elliptic Curve Diffie-Hellman function, the Salsa20 stream cipher, and the Poly1305 message-authentication code as can be seen in figure 5.

2.3.2.1 Example Let us assume that *A* and *B* want to exchange encrypted and authenticated messages. They both have generated a key pair consisting of a public and a private key. In addition, they have already exchanged their public keys over an authenticated channel. We will now go through the procedure that is required for *A* to encrypt and authenticate a message (plaintext) that can be decrypted by *B*:

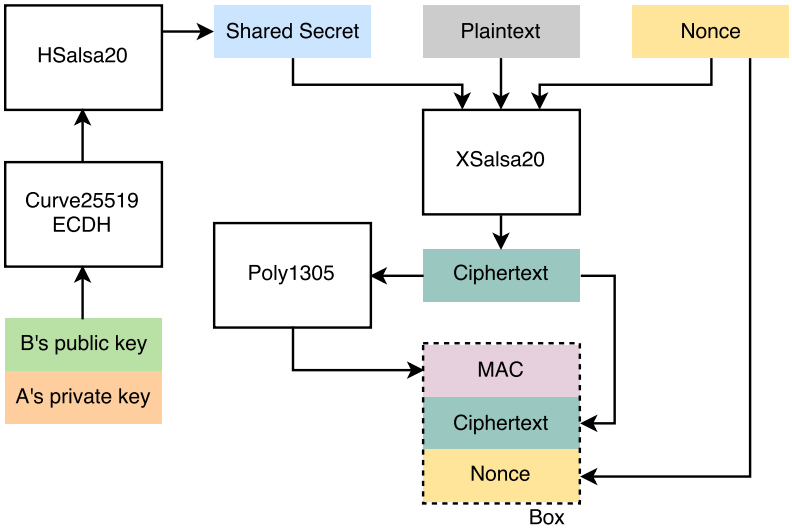

Figure 5: NaCl Public-Key Authenticated Encryption Example

1. *A* applies Curve25519 Elliptic Curve Diffie-Hellmann (ECDH) on *A*'s private and on *B*'s public key. Due to the characteristics of the elliptic curve, hashing the result with

HSalsa20 results in a 32 byte shared secret that is equal even if the keys are swapped (*B*'s private and *A*'s public key).
2. *A* generates a random or predefined 24 byte nonce. The vital part is that this nonce is only used once for the shared secret.
3. *A* uses the XSalsa20 stream cipher along with the shared secret and the nonce to encrypt the plaintext.
4. *A* applies Poly1305 on the encrypted plaintext (ciphertext) to compute a Message Authentication Code (MAC) which will be prepended to the ciphertext.
5. The MAC, the ciphertext and the nonce together form the so called *box* which can then be sent to *B*.

B can decrypt the ciphertext and verify its authenticity by reversing the steps above.

2.3.2.2 Nonces When using NaCl, it is vital to understand the role of the 24 byte nonce. The nonce is part of the Box and represents a unique message number that must never be reused for other packets that are encrypted by the same shared secret. Although a random 24 byte sequence can be used, this would open up the possibilities for replay attacks. Therefore, it is recommended to use sequence numbers along with random bytes or a timestamp-based number. To avoid replay attacks, the chosen nonce structure must also be validated by the receiver.

2.3.3 Secret-Key Authenticated Encryption

In comparison to public-key authenticated encryption, secret-key authenticated encryption does not require the derivation of the shared secret and the Curve25519 Elliptic Curve Diffie-Hellman function will be omitted. Instead, the shared secret needs to be exchanged beforehand. Other than that, the actual encryption mechanism does not differ to public-key authenticated encryption. Therefore, only the Salsa20 stream cipher, and the Poly1305 message-authentication code are being used for encryption and authentication.

Figure 5 still applies to this encryption method. The difference is that the shared secret is predefined and does not need to be derivated.

3 Security Evaluation of WebRTC

Encryption in WebRTC is mandatory. For every instance of an RTCPeerConnection, WebRTC creates a self-signed TLS certificate including a key pair for asymmetric encryption. Because

the certificates are self-signed, no explicit chain of trust exists that can be verified. In consequence, and much like the signalling channel, authentication needs to be performed by the application.

3.1 Signalling

WebRTC requires that the developer implements a signalling channel. Although encryption for WebRTC is mandatory, the *SDP* messages that need to be transferred over the signalling channel are not encrypted. Without any further encryption mechanism, they can be manipulated by intermediary nodes (e.g. at least the signalling server). Even worse, if no transport layer security is being used for the channel, every intermediary node can manipulate the signalling data.

In some cases, being able to read or even manipulate the data on the signalling server might be a requirement but we cannot think of a good reason why this should be done. ICE passwords and potentially ICE candidates could be read. Key fingerprints and ICE candidates could be changed by malicious servers in between the peers. This would open up the possibility for man-in-the-middle attacks. Confidentiality, integrity and authentication, if required, needs to be implemented by the application developer. In the browser, this is not an easy problem to solve because there is no built-in cryptography support in JavaScript and the specification of the Web Cryptography API [22] is not done, yet.

3.2 Peer Connection Authentication

The SRTP keys of the peers are being exchanged in the DTLS-SRTP negotiation, using a Diffie-Hellman based key exchange algorithm. The key fingerprint will be used to verify the public key that has been received. If the fingerprint does not match, the session will be rejected. Again, it is vital that the integrity of *offer* and *answer* messages is guaranteed. Otherwise, this mechanism cannot ensure that only the peers who exchanged *offer* and *answer* establish a peer connection.

WebRTC endpoints use SRTP for real-time and DTLS for arbitrary data. The keys required for the SRTP connections are being exchanged by the DTLS-SRTP [15] technique on the media channel. Therefore, it is independent from the signalling channel which protects the endpoints against eavesdropping on a media session.

3.3 Replay Attacks

The SRTP protocol maintains a so called *Replay List* which contains the indices of all packages that have been received in a *sliding window*. The receiver checks the index of an incoming

packet against its internal replay list. Packets with an index ahead or inside the window, but not already received, will be accepted. [17]

DTLS uses the same approach for replay detection as SRTP but the support is optional. We do not know whether or not WebRTC requires this feature.

3.4 DTLS Encryption

Because DTLS is based on TLS, it inherits nearly all of TLS's mechanisms. Over the last few years, several major attacks on TLS have been revealed [23] and we have to assume that there are more to come. Additionally, the cryptography library OpenSSL has also had several major vulnerabilities in the near past. This is relevant because a lot of WebRTC implementations will utilise OpenSSL's DTLS implementation. Fortunately, after the *Heartbleed* vulnerability has been published, a lot of effort has been made to review and improve OpenSSL. Anyhow, OpenSSL remains an extremely complex software library and we cannot rule out that there are still some major vulnerabilities that have not been discovered, yet.

3.5 Conclusion

Apart from our concerns about DTLS, the architectural design of WebRTC provides good security for the peer-to-peer connection in general. But the exposed signalling data poses a security risk, not only for unexperienced application developers. An end-to-end encrypted solution to solve this problem is not trivial.

4 SaltyRTC - Secure WebRTC based on NaCl

SaltyRTC is a software collection that has been written to set up and provide a secure WebRTC data channel even when the underlying DTLS encryption of the peer connection or the TLS encryption of the signalling implementation is completely broken. This ensures that we have a countermeasure for every possible vulnerability of DTLS or TLS.

In addition, SaltyRTC provides authentication for WebRTC peers. Moreover, SaltyRTC does not trust and does not need to trust the signalling server. Data exchanged on the signalling channel can only be read by the peers. The server has no way to manipulate data undetected. We consider this a major advantage over other signalling implementations.

All authentication is done by challenges for NaCl key pairs and a NaCl secret key that is used once. Only 64 bytes need to be exchanged over a separate channel: The public NaCl key (32 bytes) and a secret key (32 bytes) of the peer who initiates the connection. While the public key

can be exchanged over an unprotected channel, the secret key must be exchanged over a secure channel. The channel itself has to be provided by the developer. Moreover, the peers may store the public key of each others peer as a *trusted key*. In that case, further communication can be established without the need of a secure channel.

The collection consists of two libraries, one written in Java for Android and the other written in JavaScript for the browser, that provide the necessary functions to set up a peer-to-peer connection based on WebRTC. Also included in the collection is a signalling server implementation written in Python 3.

4.1 Terminology

4.1.1 WebRTC Key

A key pair that is generated by the WebRTC implementation of the browser. Neither do we have influence on the generation process of this key pair, nor can we access or modify it.

4.1.2 Permanent Key

The permanent key is a NaCl key pair for public key authenticated encryption. Each peer generates his own permanent key.

4.1.3 Authentication Token

An authentication token consists of a NaCl secret key that is valid for a single encrypted message. The token has to be exchanged over a secure channel.

4.1.4 Trusted Key

A public permanent key can be stored as a trusted key. Further communication attempts between those two peers do not require another authentication token and therefore no secure communication channel. Obviously, to make this work both peers have to store each others public permanent keys as trusted keys.

4.1.5 Session Key

The session key is a NaCl key pair which is only valid for a single peer-to-peer session. Session keys are being exchanged after the peers have authenticated each other.

4.1.6 Shared Secret

The shared secret will be derived from the public NaCl key of one peer and the private NaCl key of the other key. See the NaCl section for details. To understand the composition of the nonce, it is vital to know that both peers use the same sequence of bytes to encrypt and decrypt data.

4.1.7 Nonce

Nonces are sequences of 24 bytes that must only be used once per shared secret. A previously generated Cookie occupies the first 16 bytes of outgoing nonces. The same applies to the received cookie for incoming nonces. The last 4 bytes represent a Sequence Number to detect replay attacks. Because session keys will be used for different channels, and therefore sequence numbers will be counted separately for these channels, there are 4 random bytes called Channel Number in between the cookie and the sequence number. These random bytes are generated once and are unique for each channel that has its own sequence number counter.

Figure 6: SaltyRTC Nonce Structure

Only the authentication token uses a random nonce (e.g. 24 cryptographically secure random bytes) instead of the described format above. In this case, replay attacks can still be detected because the authentication token is only valid for a single successful decryption attempt.

4.1.8 Cookie

The cookie is being used for two things at the same time. First of all, it resembles a challenge that needs to be repeated by the other peer. The peer can thereby prove that he has the private key for the public key he transmitted. Furthermore, the cookie occupies the first 16 bytes of the 24 bytes long nonce. Each peer uses his own cookie for outgoing messages. To ensure that nonces are unique per shared secret, the peers are required to choose different cookies.

4.1.9 Channel Number

This number is unique for each channel that may be used with the same cookie but with a different sequence number counter. As part of the nonce, its only purpose is to ensure that no nonce is being used more than once with the same shared secret.

4.1.10 Sequence Number

The sequence number plays a vital role as part of the nonce. It is mainly used to detect replay attacks. Because the sequence number will always be incremented with each packet sent on its channel, it also ensures that no nonce is being used repeatedly.

4.1.11 Initiator

The initiator initiates the peer connection. In case the initiator has not stored a previously generated private permanent key and a trusted key of the other peer, a new permanent key will be generated alongside an authentication token.

4.1.12 Responder

The responder receives information from the initiator. This information could just be a wakeup call in case the initiator is already trusted or contain the public permanent key of the initiator including an authentication token.

4.1.13 Signalling Channel Path

As the signalling channel has been implemented using WebSocket, we can specify our own paths to separate incoming connections. A path is a simple ASCII string and consists of the hex value of the initiators public permanent key. Initiator and responder connect to the same path.

4.2 Architecture

In figure 7, we have illustrated the architecture of SaltyRTC. Before SaltyRTC will be explained in detail, we will provide a basic overview of each section from top to bottom:

We have used the *Web API* to develop the browser version of SaltyRTC and the native Android library (which is not visible in figure 1 but basically wraps the C++ API) to build the Android version of SaltyRTC. While the APIs are not that far apart from each other, the languages, **Java** and **JavaScript**, are not alike, so two more or less similar library versions had to be developed.

SaltyRTC is divided into three major sections: The **Signalling Channel**, the **Peer Connection** and the **Data Channel**. All three sections require packet **validation** and a **cryptography** interface to encrypt and decrypt messages. Additionally, we are able to separate SaltyRTC **session**s from each other and provide high-level **event**s and **state**s. The events and states of the three major sections have been unified to simplify the usage of SaltyRTC. However, developers are

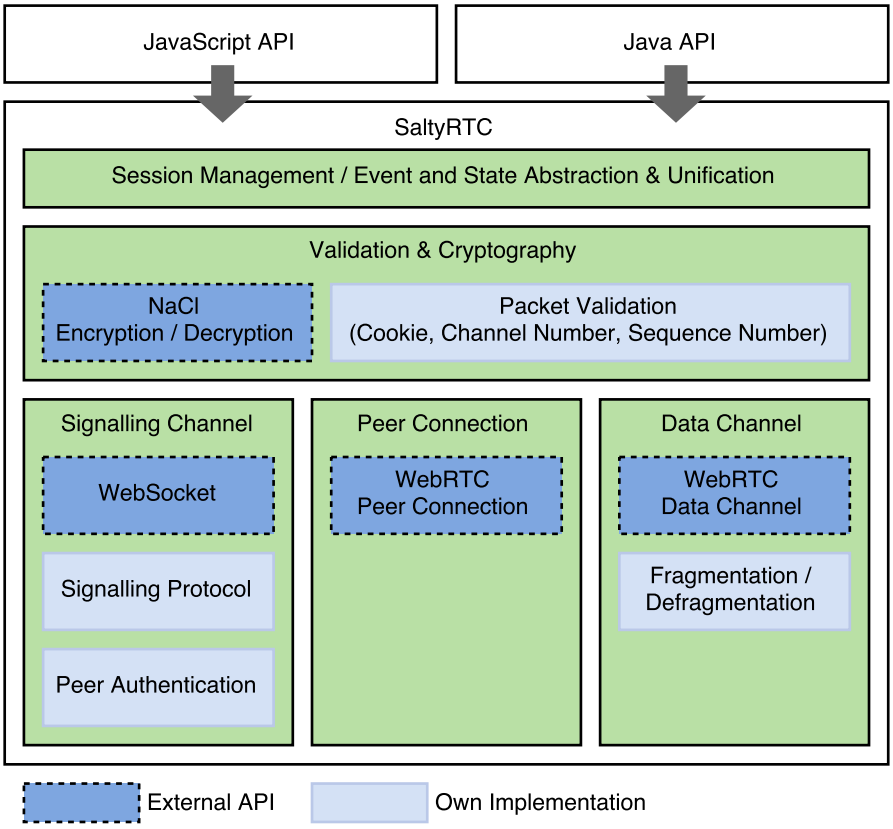

Figure 7: SaltyRTC Architecture

still able to access the low-level events and states of the signalling channel, the peer connection and the data channel.

The **Signalling Channel** uses WebSocket and a custom binary signalling protocol to authenticate the other peer. After the communication partner has been authenticated, the **Peer Connection** section provides the required metadata to setup a peer-to-peer connection. As soon as the peer-to-peer connection has been established, the **Data Channel** can be used to exchange arbitrary data.

4.3 Exchanging the Authentication Token and the Public Key

When two peers want to connect to each other for the first time, the initiator must generate an authentication token. This token needs to be transmitted to the other peer alongside the public permanent key of the initiator. In the section Authentication of the Peers, the authentication token will be used by the responder.

It is vital to understand that SaltyRTC does not declare how the authentication token and the public permanent key of the initiator need to be transmitted. However, we have minimised the amount of data to 64 bytes which can easily be encoded into a QR-Code.

4.4 Signalling Channel

As mentioned before, WebRTC does not provide a signalling solution but leaves this task up to the developer. SaltyRTC provides such a signalling channel implementation based on WebSocket with TLS encryption.

The signalling protocol has been designed in a way that the channel itself does not need to be trusted. The encrypted data is simply relayed to the other peer and can only be read by either of the peers.

4.4.1 Packet Structure

In the following, we will describe and visualise the structure of the payload of the WebSocket packet:

1. A single byte that indicates the receiver or sender of the packet, depending on whether the packet is being sent (*to*) or has been received (*from*). The receiver byte will be explained in the table below. This byte is not encrypted.

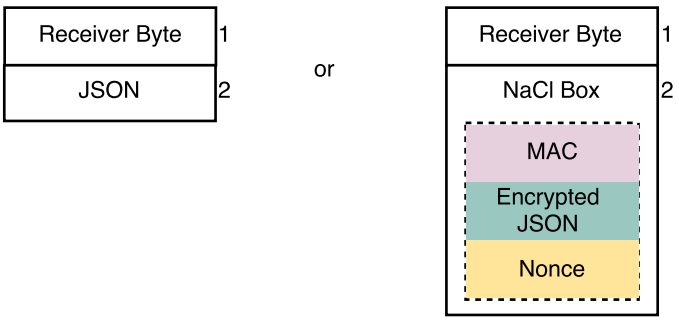

Figure 8: Signalling Channel Packet Structure

2. A serialised JSON object or a NaCl Box containing a serialised JSON object. The nonce of this box complies to the format described in the Nonce section. Which case applies is known to all parties at any time because the plaintext serialised JSON object is only needed during the handshake.

Receiver	**Value**
from/to Server	0x00
from/to Initiator	0x01
from/to Responder #1	0x02
from/to Responder #2	0x03
...	...

Table 3: Receiver Types and Values

With the receiver byte, the server and the peers can determine which key has to be used to decrypt the NaCl box and how to validate the nonce. In addition, the server requires this byte to determine whether the packet needs to be relayed to the other peer or is directed at the server. Before the authentication of a specific client is complete, the only receiver the client may use is *Server*. In addition, *Responders* are not allowed to communicate with other *Responders*.

The receivers *Server* and *Initiator* always have the same value. Because there can be multiple *Responders* connected to the same signalling channel at the same time, each successful authentication of a responder towards the server requires an identifier. Therefore, the maximum amount of connected clients on a path is 255.

4.4.2 Packet Types: Client-to-Server

This section describes the various packet types that can be exchanged between server and client (e.g. the receiver byte is set to *from/to Server*). We will describe and provide an example for each packet type. The packets are UTF-8 encoded JSON objects.

4.4.2.1 server-hello This is the first message that occurs on the signalling channel. The server generates and sends his public key alongside a hex encoded cookie to the client.

Encryption: None (apart from the underlying TLS layer)

```
{
   "type": "server-hello",
   "key": "debc3a6c9a630f27eae6bc3fd962925bdeb63844c09103f609bf7082bc383610",
   "m-cookie": "af354da383bba00507fa8f289a20308a"
}
```

4.4.2.2 client-hello In case that the client is the responder, the client will answer to a server-hello with his hex encoded public permanent key. The initiator will not send this packet type to the server. Thereby, the server can differentiate between initiator and responder.

Encryption: None (apart from the underlying TLS layer)

```
{
   "type": "client-hello",
   "key": "55e7dd57a01974ca31b6e588909b7b501cdc7694f21b930abb1600241b2ddb27"
}
```

4.4.2.3 client-auth Both initiator and responder send this packet type. It contains the repeated cookie (*y-cookie*) that the server sent along with the server-hello and a hex encoded cookie the client generates (*m-cookie*).

Encryption: NaCl Box (Server's Session Public Key, Client's Permanent Private Key)

```
{
   "type": "client-auth",
   "y-cookie": "af354da383bba00507fa8f289a20308a",
   "m-cookie": "18b96fd5a151eae23e8b5a1aed2fe30d"
}
```

4.4.2.4 server-auth To complete the authentication process, the server repeats the cookie that the client sent in the client-auth packet. The additional field *responders* contains a list of hex encoded identities of responders that have authenticated themselves towards the server. Both initiator and responder will receive this packet. But the *responders* field will only be included in the packet that is intended for the initiator.

Encryption: NaCl Box (Server's Session Private Key, Client's Permanent Public Key)

```
{
   "type": "server-auth",
   "y-cookie": "18b96fd5a151eae23e8b5a1aed2fe30d",
   "responders": [
      "02",
      "03"
   ]
}
```

4.4.2.5 new-responder When a new responder has completed the server's authentication process and an initiator is connected, the server will send this message to the initiator. It contains the hex encoded identity (*id*) of the newly connected responder.

Encryption: NaCl Box (Server's Session Private Key, Client's Permanent Public Key)

```
{
   "type": "new-responder",
   "id": "04"
}
```

4.4.2.6 drop-responder After the initiator is authenticated towards the server, he may request that one or more responders shall be dropped from the server. The *id* field contains the hex encoded identity of a responder.

Encryption: NaCl Box (Server's Session Private Key, Client's Permanent Public Key)

```
{
   "type": "drop-responder",
   "id": "02"
}
```

4.4.2.7 send-error In case that the server could not relay a message from one peer to the other peer, the server will send this message to the client who originally sent the message. The original message will be hex encoded and included in the *message* field.

Encryption: NaCl Box (Server's Session Private Key, Client's Permanent Public Key)

```
{
   "type": "send-error",
   "message": "..."
}
```

4.4.3 Packet Types: Peer-to-Peer

This section describes the various packet types that can be exchanged between an initiator and a responder (e.g. the receiver byte is set to *from/to Initiator* or *from/to Responder #x*). We will provide and describe an example for each packet type. The packets are UTF-8 encoded JSON objects.

Note: The packet payload cannot be decrypted by the server because different NaCl keys will be used.

4.4.3.1 token The responder sends his hex encoded public permanent key to the initiator. In case that both peers have stored each other's permanent keys as trusted keys, this packet will be skipped.

Encryption: NaCl Box (Authentication Token)

```
{
   "type": "token",
   "key": "55e7dd57a01974ca31b6e588909b7b501cdc7694f21b930abb1600241b2ddb27"
}
```

4.4.3.2 key The peer announces his hex encoded public session key accompanied by a hex encoded random cookie. Both peers need to send this packet.

Encryption: NaCl Box (Sender's Private Permanent Key, Receiver's Public Permanent Key)

```
{
   "type": "key",
   "key": "bbbf470d283a9a4a0828e3fb86340fcbd19efe75f63a2e51ad0b16d20c3a0c02",
   "m-cookie": "957c92f0feb9bae1b37cb7e0d9989073"
}
```

4.4.3.3 auth The peer who received a previously sent key packet, repeats the received cookie. Both peers need to send this packet.

Encryption: NaCl Box (Sender's Private Session Key, Receiver's Public Session Key)

```
{
   "type": "auth",
   "y-cookie": "957c92f0feb9bae1b37cb7e0d9989073",
}
```

4.4.3.4 offer The initiator sends the WebRTC *offer* SDP message in the *data* field along with a *session*. This *session* is a random 32 printable character string that needs to be provided in further messages from both peers. With this field, the WebRTC session is identifiable.

Encryption: NaCl Box (Sender's Private Session Key, Receiver's Public Session Key)

```
{
   "type": "offer",
   "session": "QFyatn4rwnJSPiOOru8JTAo7nM2syOWs",
   "data": "..."
}
```

4.4.3.5 answer As soon as the responder received and processed an offer packet, he sends the WebRTC *answer* SDP message in the *data* field to the initiator.

Encryption: NaCl Box (Sender's Private Session Key, Receiver's Public Session Key)

```
{
   "type": "answer",
   "session": "QFyatn4rwnJSPiOOru8JTAo7nM2syOWs",
   "data": "..."
}
```

4.4.3.6 candidate Both peers may send WebRTC *ICE candidates* at any time after offer and answer messages have been exchanged. The *candidates* will be provided in the *data* field.

Encryption: NaCl Box (Sender's Private Session Key, Receiver's Public Session Key)

```
{
   "type": "candidate",
   "session": "QFyatn4rwnJSPiOOru8JTAo7nM2syOWs",
   "data": "..."
}
```

4.4.3.7 restart Initiator and responder may request that a WebRTC session shall be restarted. This packet type can be used after the session keys have been exchanged and the cookies have been repeated.

Encryption: NaCl Box (Sender's Private Session Key, Receiver's Public Session Key)

```
{
   "type": "restart"
}
```

4.4.4 Connection Build-Up

Both initiator and responder connect to the same signalling channel path which consists of the initiator's hex encoded public permanent key. Therefore, the risk of a path collision is as negligible as the risk of generating the same private key twice.

Because we use WebSocket with TLS, the connection build-up includes a TCP, TLS and a WebSocket handshake.

4.4.5 Authentication towards the Server

When a peer is connected to the signalling server, he needs to authenticate himself towards the server. The procedure differs for initiator and responder as can be seen in figure 9.

The grey dotted arrow in figure 9 indicates that the TCP, TLS and the WebSocket handshake need to be done beforehand, including transmitting the signalling channel path.

A green arrow indicates that the message payloads are NaCl public-key encrypted between server and initiator/responder. Black arrows mark unencrypted messages (although this is a bit misleading because these messages are still TLS encrypted).

4.4.5.1 Initiator

1. The server generates and sends his public key alongside a cookie to the initiator.
2. Because both public keys have been exchanged, the initiator can now repeat the server's cookie and send his own cookie in an encrypted message.
3. To complete the authentication process, the server repeats the initiator's cookie.

The initiator does not need to send his public permanent key because the key has already been provided as the WebSocket path. Therefore, the initiator can directly send NaCl public-key encrypted payloads while the responder still needs to send his public permanent key. As the

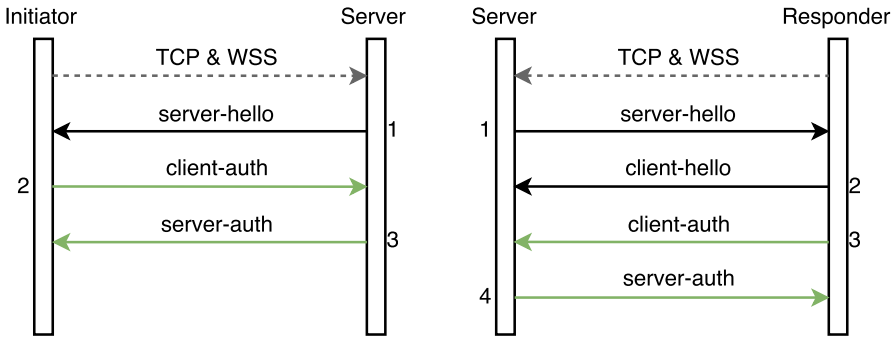

Figure 9: SaltyRTC Signalling Channel Server Authentication

initiator has the privilege to disconnect responders from the signalling channel, this authentication procedure proofs towards the server that the initiator has the private key to the path he connected to.

4.4.5.2 Responder

1. The server generates and sends his public key alongside a cookie to the responder.
2. To be able to send encrypted messages, the responder sends his permanent public key to the server.
3. Now, that the responder has sent his public key to the server, he can repeat the server's cookie and send his own cookie in an encrypted message.
4. To complete the authentication process, the server repeats the responder's cookie.

4.4.6 Authentication of the Peers

As soon as initiator and responder have authenticated themselves towards the server on the same path, the connections are linked together. Both peers may send payloads to each other and the signalling server simply relays them to the other peer.

Now, the peers need to authenticate themselves towards each other and announce their session keys. This procedure can be seen in figure 10.

The orange arrow in the figure expresses a payload that is encrypted by the authentication token. A green arrow indicates that the message payloads are encrypted by the peers permanent keys. A blue arrow marks message payloads that have been encrypted with the peers session keys.

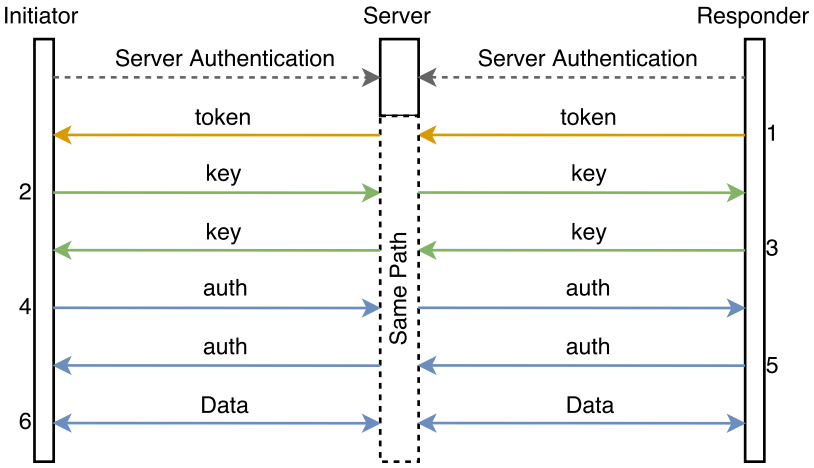

Figure 10: SaltyRTC Signalling Channel Peer Authentication

1. The responder starts by sending his public permanent key encrypted by the authentication token to the initiator. In case that both peers have stored each other's permanent keys as trusted keys, this step will be skipped and no authentication token is required.
2. A session key will be generated and sent by the initiator, accompanied by a cookie.
3. Just like the initiator, the responder will also send a generated session key and a cookie.
4. To mitigate replay attacks, the initiator now repeats the responder's cookie.
5. And the responder repeats the initiator's cookie.
6. Both parties may now send arbitrary data to each other, encrypted by the session keys.

Not shown in figure 10 is the communication between initiator and server. The initiator may request that formerly connected responders shall be disconnected from the signalling channel path when a new responder has authenticated himself.

4.4.7 Exchanging Offer, Answer and Candidates

After both authentication procedures are complete, the peers are free to send arbitrary data to each other over the signalling channel. The initiator will start sending an *offer* message and the responder will answer with an *answer* message. Finally, the ICE candidates will be exchanged and the peer-to-peer connection can be established. All these messages have been described in the WebRTC section and are encrypted with the established session keys.

4.5 Data Channel

When the WebRTC peer connection has been established, initiator or responder may create WebRTC data channels. To use SaltyRTC's encryption mechanisms, the RTCDataChannel object can be supplied to a function of the library which wraps the RTCDataChannel object. Both peers have to wrap their RTCDataChannel objects. The result is a RTCDataChannel-like object that can be used to send arbitrary data where encrypting and fragmenting will happen automatically. For encryption, the previously established session keys will be used.

4.5.1 Packet Structure

In the following, we will describe and visualise the structure of the payload of the data channel packet:

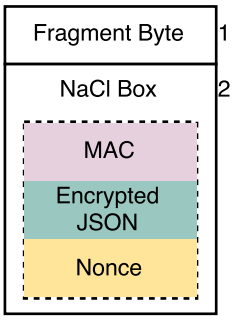

Figure 11: Data Channel Packet Structure

1. A single byte that indicates whether there are more fragments (any value but 0x00) or the message is complete. This byte is not encrypted.
2. A part of or a whole NaCl Box containing arbitrary data. The nonce of this box complies to the format described in the Nonce section. Fragmented boxes will be reassembled when they are complete.

Note: Currently, the WebRTC library from Google only supports sending messages up to 16 KB. [24] To avoid data loss, messages greater than 16 KB have to be fragmented. Once this issue is resolved, the fragment byte will be removed. A negative side effect of this temporary and very simple fragmenting solution is that, until the fragment byte is removed, SaltyRTC only supports ordered and reliable data channels.

4.6 Peer-to-Peer Connection Build-Up

As soon as the RTCPeerConnection is being created, the initiator acquires his permanent key. Depending on whether the responder is already trusted or not, the public permanent key and an authentication token may need to be transmitted to the responder. The initiator connects to the signalling channel path and authenticates himself towards the signalling server.

Once the responder has received the information, he also connects to the same signalling channel path. In case the responder has received an authentication token, he will send his public permanent key encrypted by the authentication token. Now that the responder is authenticated, both peers generate and send each other their session keys along with a random cookie. Each peer repeats the received cookie encrypted by the session keys. As soon as the peers have validated that the repeated cookie matches the sent cookie, they are ready to transmit arbitrary data over the signalling channel.

This is the time where all signalling data from WebRTC, such as *offer*, *answer* and *candidates*, will be transmitted over the signalling channel.

Once the candidate gathering is complete, the peer connection will be established. The developer may now create WebRTC data channels and use SaltyRTC to send data directly to the other peer encrypted by the session key.

4.7 Security Analysis

4.7.1 Preconditions

Before we can talk about various attack scenarios, we need to make a few assumptions for both communication partners:

- The operating system of the device and the running applications are not compromised.
- The browser, e.g. the JavaScript sandbox is not compromised.
- The Android application is not compromised.

4.7.2 Attack Vectors

4.7.2.1 Authentication Token Although SaltyRTC does not declare how an authentication token should be exchanged, we can assume that using QR codes will be a commonly used and convenient way of exchanging the token. Obviously, exchanging this token in a public place is a bad idea and it is probably impossible to eliminate the possibility of eavesdropping. Even at home, wiretapping is still possible if the rooms of the residence have been compromised, for example by an intelligence agency. Nonetheless, the frequency of such attacks can be reduced significantly by using the *trusted key* feature of SaltyRTC.

4.7.2.2 Signalling Channel In this scenario, we assume that the signalling server has been compromised. But because SaltyRTC does not trust the signalling server in the first place, the only attack that can be made is a denial of service. Still, this is a relevant attack because no WebRTC peer connection can be established.

4.7.3 Protection Mechanisms

The following section summarises the most vital security mechanisms of SaltyRTC and which attacks they prevent.

4.7.3.1 Authentication When the authentication token has been exchanged in a secure manner, both peers can assure authentication of each other.

The initiator either authenticates the responder by receiving the responder's public permanent key via the authentication token, or he already knows the public permanent key of the responder. For both cases, only the initiator and the responder have the shared secret that can decrypt messages.

The other peer, the responder, also knows the public permanent key of the initiator before he connects to the signalling server. Again, only the initiator and the responder have the shared secret to decrypt messages.

To summarise, only initiator and responder are able to communicate with each other. An attacker would not be able to derive the necessary shared secret from the messages that are being exchanged on the signalling and the data channel.

4.7.3.2 Protection against Denial of Service on the Signalling Channel Because of the receiver byte (see Packet Structure for details), the amount of responders connected to a signalling channel path is limited to 254. If an attacker knows the public permanent key of the initiator, he can authenticate 254 responders towards the signalling server and hinder further responders from authenticating on the same signalling path who want to communicate with the initiator.

However, the server will announce all authenticated responders at the beginning and will continue to announce each newly authenticated responder to the initiator. The initiator has the privilege to request that a responder shall be dropped from the server. A timeout is being used for each responder that has not authenticated himself towards the initiator. When the timeout expires, the initiator will request that the responder shall be dropped.

4.7.3.3 Protection against Replay Attacks The Nonce consists of a random 16 byte cookie a 4 byte channel number and a 4 byte sequence number. A peer who receives an encrypted

message knows the cookie of the sender because the sender has announced the cookie he will use during the authentication process. Channel number and sequence number can only be incremented. Therefore, a repeated message can be detected easily by verifying the cookie and the sequence number for the channel.

For the data channel: The attacker would have to break the replay detection or the encryption of DTLS first before this mechanism even takes place.

4.7.3.4 Uniqueness of Nonces Because we use NaCl for encryption, a Nonce is required to be unique per shared secret. Part of the nonce is a 16 byte cookie that must be a cryptographically secure sequence of random bytes. The chance of generating a cookie twice is negligible and the attacker would have to do the whole handshake process of the signalling server and wait until the victim sends the first message that includes a newly generated cookie. However, it is also possible to reduce the attacker's chance of success significantly by enforcing a timeout after each failed authentication attempt.

4.7.3.5 Maintaining Ciphertext Integrity The Message Authentication Code (MAC) of the NaCl Box ensures that messages are authenticated.

4.7.3.6 Forward Secrecy In case that the attacker managed to break the DTLS encryption of the WebRTC peer connection and obtain the private permanent key of one of the peers, the messages of a session's data channel still cannot be decrypted. The attacker would also need to obtain the private session key from one of the peers. However, the session keys are only valid for a limited period of time and will not be stored permanently by either of the peers. This mechanism ensures that compromised private permanent keys do not compromise encrypted data of previous sessions.

4.8 Possible Improvements

In this section, we will enumerate a list of several improvements that could be applied to the SaltyRTC software collection.

4.8.1 Unordered and Partially Reliable Delivery Support

Due to the fact that the WebRTC library from Google currently does not reliably transfer messages greater than 16 KB [24], we had to implement manual message fragmentation. The first byte of each message indicates whether there are more upcoming fragments or not. Therefore, the underlying data channel must be reliable and ordered. Because this is a temporary

problem that will likely be resolved by Google in the near future, no big effort has been made to work around it.

A sliding window approach for the sequence number validation could be used when fragmenting is not needed anymore.

4.8.2 MessagePack instead of JSON

The usage of MessagePack instead of JSON objects would reduce the packet size of many messages that include binary data. JSON requires that binary data is encoded into printable characters. MessagePack allows for binary data to be packed directly. However, there seems to be no officially developed JavaScript library which is the main reason why it has not been used for the protocol design at the first place.

4.8.3 Push Message Extension

After an authentication token has been set on the server, the server may send a push message to the other peer. This tells the other peer to connect to the signalling channel in case the received key in the push message is a trusted key.

4.8.4 Audio and Video

Currently, SaltyRTC does not handle audio or video. Simply, because we did not need it, yet. Additionally, we do not know whether the media API of HTML5 supports modifying the media stream data and whether or not that would break the various techniques of the codecs, such as echo cancellation for voice data. On the sender's side, we need to be able to modify the stream data **after** it has been processed by the codecs. And on the receiver's side, the same applies to the stream data **before** it has been processed by the codecs. Both pre- and post-processing are required.

However, you can still use SaltyRTC to set up the peer connection in a secure manner and then use the WebRTC API for audio and video.

5 Threema Web Client Prototype

Threema is a mobile messenger application that provides end-to-end encryption. [25] It has been developed by the *Threema GmbH* which is based in Pfäffikon, Switzerland.

5.1 Introduction

Because of the end-to-end encryption, the Threema server only stores messages momentarily until they have been transferred to the receiver. Therefore, the chat history, contacts, etc. are only stored on a single device; Threema has no multi-device support. A user who owns multiple devices and wants to see her/his chat history on both of them, faces a problem which has not been resolved until now.

5.2 Analysis

Using the same Threema ID on multiple devices does not work because the server only allows one connection from the same Threema ID at the same time. Moreover, messages from contacts would only be transferred to one of the devices but not both. Clearly, this is not what we want.

To provide multi-device support, we have to provide a way to retrieve data from the device which has the Threema ID and its private key. From now on, we will call this device the *Threema node* or just *node*.

We estimate that most Threema users carry the node with them most of the time. But to write long messages, a notebook is much more convenient. In those cases, the users are probably even connected to the same wireless access point on both devices. Thus, transferring the data from the node to the notebook over the Internet would be pointless. A direct connection over the local area network is preferable.

Other Threema users may have a device that they leave at home all or most of the time. If this device is the Threema node, they must also be able to access the device from other locations over the Internet.

Most importantly, the client needs to be authenticated towards the node and the data needs to be transferred securely. Only the node and the connecting client shall be able to decrypt the data they exchange.

What we want is a direct and secure way to communicate with the node from other devices over the shortest possible route.

5.3 Solution

SaltyRTC is the answer. It does exactly what we require: A direct and secure way to communicate with the node over the shortest possible route. Clients, who want to connect to the node, have to be authenticated explicitly by the node by using QR codes that resembles the authentication token. A node can trust the permanent key of a client and further communication

attempts do not require another authentication token. When a client is trusted by the node, a simple push message sent to the node suffices and the node can start connecting to the client.

We have developed a protocol on top of SaltyRTC's data channel which provides access to contacts, messages and various other data. Of course, it is also possible to send messages to Threema contacts on the client who connects to the node. The idea is that the client is able to do everything the Threema application can do as well.

5.4 Prototype

A prototype has been developed for the browser which uses SaltyRTC to connect to the Threema node. We are able to retrieve contacts, groups and their avatars, display conversations and their chat history, receive and handle new messages, contacts, etc. and, most importantly, are able to send Threema messages from the client who connects to the node.

Both devices do not have to be on the same network. But because ICE tries to use the *best* route, the devices will most likely gain a latency and throughput benefit if they are on the same network.

For the prototype, we have not used the permanent key feature. A new permanent key pair will be generated for each session. The browser generates and shows the authentication token in form of a QR code which the Android app has to scan for every new session. This could be changed over to a push notification in the future (see the Push Message Extension section for details).

5.4.1 Communication

The browser takes the role of the SaltyRTC initiator and the Threema Android app the role of the responder.

1. First of all, the browser creates a SaltyRTC instance which automatically generates a new permanent key pair. The browser requests an authentication token and displays the hex encoded token in a QR code.
2. In the app, the user has to start the *Web Client*. A SaltyRTC instance starts and a new permanent key pair will be generated. The app will show a camera preview and the user has to scan the QR code with the smartphone camera.
3. Both browser and app connect to the same signalling server and use the permanent public key of the browser as the *path* value.
4. The app uses the secret key provided in the authentication token to authenticate itself towards the browser.

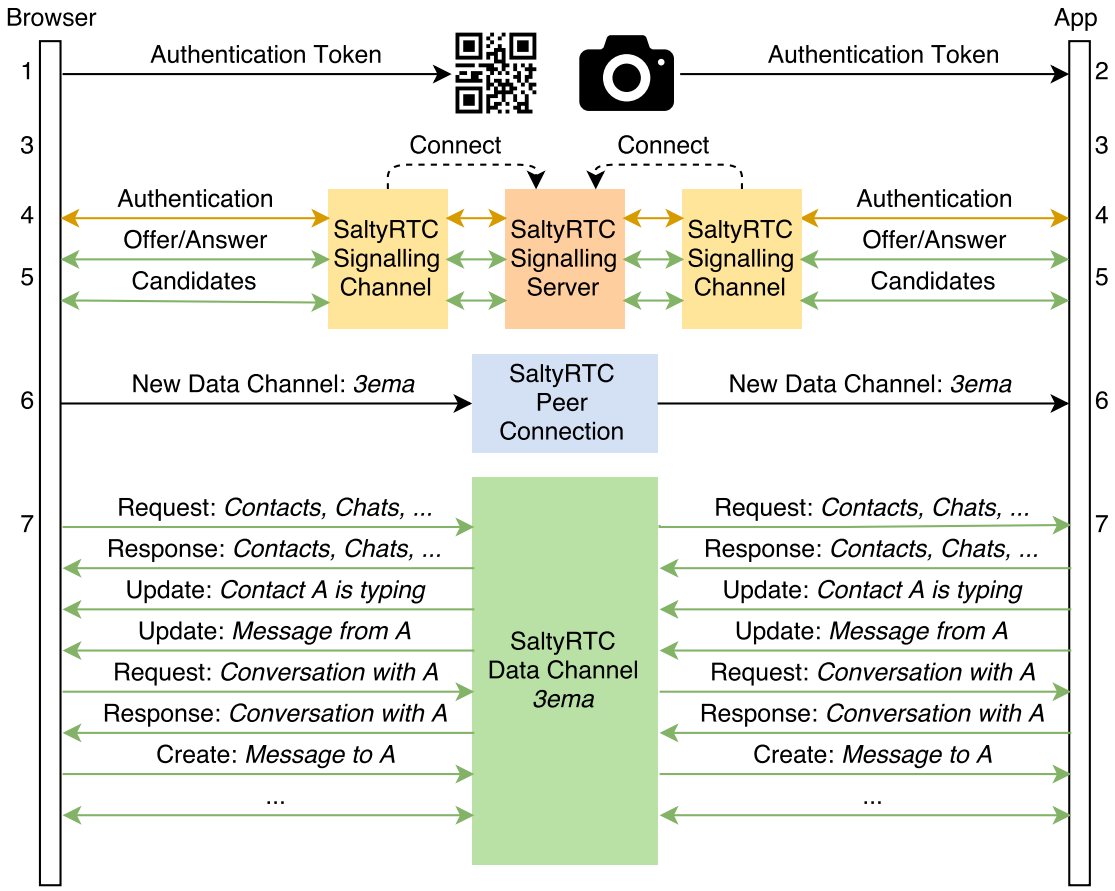

Figure 12: Threema Web Client Communication

5. To establish the peer connection, the browser sends the *offer* message and the app responds with an *answer* message. Afterwards, ICE candidates will be exchanged.
6. The peer-to-peer connection is being established and the browser creates a data channel with the label *3ema*. The app receives information about the newly created data channel and waits for incoming messages.
7. Now, the browser requests various initial data, e.g. the contact list, a list of conversations, the avatar images, etc. and the app sends the requested data back to the browser. The browser will receive *updates* to requested data and can request additional data from the app.

5.4.2 Screenshot

In figure 13 we can see the contact list on the left side and a conversation on the right side.

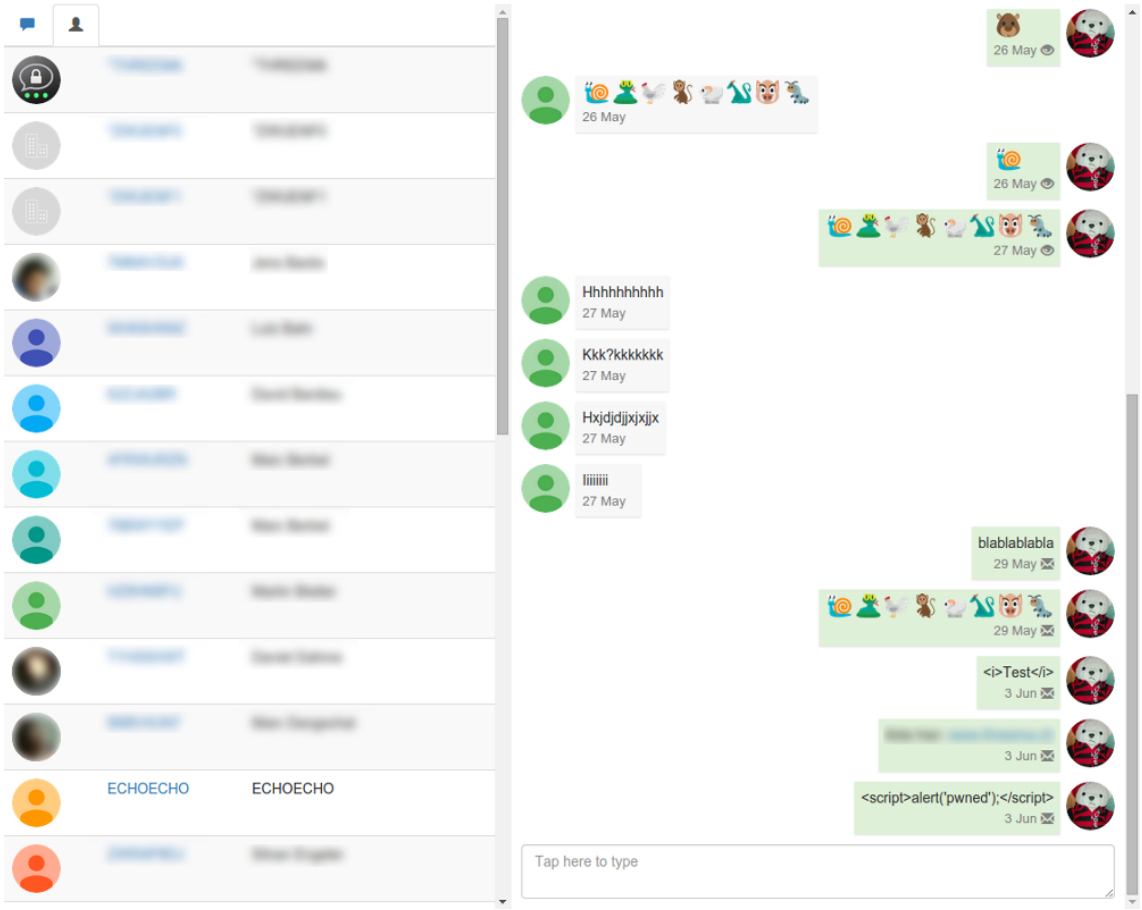

Figure 13: Threema Web Client Prototype

5.5 Conclusion

The prototype shows a typical use case for SaltyRTC. It also proves that multi-device support for Threema is tricky but definitely possible. The only constraint left is that the node device has to be online and reachable over the Internet.

6 References

[1] RFC 5245 - Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols, 2010. *https://tools.ietf.org/html/rfc5245*. Accessed: 2015-08-06.

[2] RFC 6455 - The WebSocket Protocol, 2011. *https://tools.ietf.org/html/rfc6455*. Accessed: 2015-08-06.

[3] The WebSocket API, 2014. *https://w3c.github.io/websockets/*. Accessed: 2015-08-06.

[4] WebRTC 1.0: Real-time Communication Between Browsers, *http://www.w3.org/TR/2015/WD-webrtc-20150210/*. Accessed: 2015-10-18.

[5] WebRTC Architecture, *http://www.webrtc.org/architecture*. Accessed: 2015-08-06.

[6] webrtc/adapter, *https://github.com/webrtc/adapter*. Accessed: 2015-08-10.

[7] Is WebRTC ready yet?, *http://iswebrtcreadyyet.com*. Accessed: 2015-09-07.

[8] RFC 768 - User Datagram Protocol, 1980. *https://tools.ietf.org/html/rfc768*. Accessed: 2015-08-14.

[9] RFC 5389 - Session Traversal Utilities for NAT (STUN), 2008. *https://tools.ietf.org/html/rfc5389*. Accessed: 2015-08-14.

[10] Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN), 2010. *https://tools.ietf.org/html/rfc5766*. Accessed: 2015-08-24.

[11] RFC 6156 - Traversal Using Relays around NAT (TURN) Extension for IPv6, 2011. *https://tools.ietf.org/html/rfc6156*. Accessed: 2015-08-24.

[12] RFC 4566 - SDP: Session Description Protocol, 2006. *https://tools.ietf.org/html/rfc4566*. Accessed: 2015-08-25.

[13] RFC 3264 - An Offer/Answer Model with the Session Description Protocol (SDP), 2002. *https://tools.ietf.org/html/rfc3264*. Accessed: 2015-08-25.

[14] RFC 6347 - Datagram Transport Layer Security Version 1.2, 2012. *https://tools.ietf.org/html/rfc6347*. Accessed: 2015-08-29.

[15] RFC 5764 - Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP), 2010. *https://tools.ietf.org/html/rfc5764*. Accessed: 2015-08-29.

[16] RFC 3550 - RTP: A Transport Protocol for Real-Time Applications, 2003. *https://tools.ietf.org/html/rfc3550*. Accessed: 2015-08-27.

[17] RFC 3711 - The Secure Real-time Transport Protocol (SRTP), 2004. *https://tools.ietf.org/html/rfc3711*. Accessed: 2015-08-26.

[18] RFC 4960 - Stream Control Transmission Protocol, 2007. *https://tools.ietf.org/html/rfc4960*. Accessed: 2015-09-04.

[19] Grigorik, I. 2013. *High Performance Browser Networking*. O'Reilly Media.

[20] Internet-Draft - Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol, 2015. *https://tools.ietf.org/html/draft-ietf-mmusic-trickle-ice-02*. Accessed: 2015-10-15.

[21] Bernstein, D.J. Cryptography in NaCl.

[22] Web Cryptography API, 2014. *http://www.w3.org/TR/WebCryptoAPI/*. Accessed: 2015-10-07.

[23] RFC 7457 - Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS), *https://tools.ietf.org/html/rfc7457*. Accessed: 2015-10-16.

[24] WebRTC - Chrome - FAQ about the current implementation, *http://www.webrtc.org/web-apis/chrome*. Accessed: 2015-10-15.

[25] Threema - Seriously secure messaging., *https://threema.ch*. Accessed: 2015-10-15.