Tuukka Koistinen

Implementation and Evaluation of Security on a Gateway for Web-based Real-Time Communication

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 22.5.2014

Thesis supervisor:

Prof. Jörg Ott

Thesis advisor:

M.Sc. (Tech.) Tuomas Erke

Aalto University
School of Electrical Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Tuukka Koistinen

Title: Implementation and Evaluation of Security on a Gateway for Web-based Real-Time Communication

Date: 22.5.2014 Language: English Number of pages:8+65

Department of Communications and Networking

Professorship: Networking Technology Code: S-38

Supervisor: Prof. Jörg Ott

Advisor: M.Sc. (Tech.) Tuomas Erke

Web Real-Time Communication (WebRTC) is a set of standards that are being developed, aiming to provide native peer-to-peer multimedia communication between browsers. The standards specify the requirements for browsers, including a JavaScript Application Programming Interface (API) for web developers, as well as the media plane protocols to be used for connection establishment, media transportation and data encryption.

In this thesis, support for Interactive Connectivity Establishment (ICE) and media encryption was implemented to an existing gateway prototype. The gateway was originally developed to connect the novel WebRTC possibilities with existing IP Multimedia Subsystem (IMS) services, but it was lacking the necessary security functionalities. The performance of the gateway was measured and analyzed in different call scenarios between WebRTC clients. Two key elements, CPU load of the gateway and packet delay, were considered in the analysis.

In addition to single call scenarios, the tests included relaying of ten simultaneous HD video calls, and relaying of ten simultaneous audio calls. Estimates based on the measurements suggest, that the overall capacity of a single gateway between two WebRTC clients ranges from 14 simultaneous HD video calls to 74 simultaneous audio calls. The median delay in the gateway remained under 0.2 milliseconds throughout the testing.

Keywords: WebRTC, real-time, multimedia, gateway, peer-to-peer, JavaScript, encryption, IMS, ICE

AALTO-YLIOPISTO	DIPLOMITYÖN
SÄHKÖTEKNIIKAN KORKEAKOULU	TIIVISTELMÄ

Tekijä: Tuukka Koistinen

Työn nimi: Tietoturvan toteutus ja arviointi verkkopohjaiseen reaaliaikaiseen kommunikointiin tarkoitetussa yhdyskäytävässä

Päivämäärä: 22.5.2014 Kieli: Englanti Sivumäärä:8+65

Tietoliikenne- ja tietoverkkotekniikan laitos

Professuuri: Tietoverkkotekniikka Koodi: S-38

Valvoja: Prof. Jörg Ott

Ohjaaja: DI Tuomas Erke

Verkkopohjainen reaaliaikainen kommunikointi (WebRTC) on joukko uusia standardeja, jotka mahdollistavat selainten välisen multimediakommunikoinnin. Nämä standardit määrittelevät vaatimukset selaimille, sisältäen JavaScript-ohjelmointirajapinnan sovelluskehittäjille, kuin myös mediatason protokollat, joita käytetään yhteyden muodostamiseen, median välittämiseen sekä tiedon salaukseen.

Tuki interaktiiviselle yhteyden luomiselle (ICE) ja tiedon salaukselle toteutettiin olemassaolevalle yhdyskäytäväprototyypille. Kyseinen yhdyskäytävä oli alunperin luotu yhdistämään WebRTC-mahdollisuudet olemassaolevaan IP-pohjaiseen multimediaverkkoon, mutta siitä puuttui tarvittavat tietoturvaominaisuudet. Yhdyskäytävän suorituskyky mitattiin ja analysoitiin eri puhelutyypeillä WebRTC-käyttäjien välillä. Analyysi keskittyi kahteen suureeseen: yhdyskäytävän prosessointikuorma sekä pakettien viive.

Yksittäisten puheluiden lisäksi yhdyskäytävää kuormitettiin kymmenellä HD videopuhelulla ja kymmenellä audiopuhelulla. Mittausten perusteella tehtyjen arvioiden mukaan kahden WebRTC-käyttäjän välillä olevan yksittäisen yhdyskäytävän suorituskyky yltää 14:stä yhtäaikaisesta HD videopuhelusta 74:ään yhtäaikaiseen audiopuheluun. Mediaaniviive pysyi kaikissa testeissä alle 0.2 millisekunnissa.

Avainsanat: WebRTC, reaaliaikainen, multimedia, yhdyskäytävä, JavaScript, tietoturva, IMS, ICE

Preface

This Master's thesis has been done at Ericsson Finland, as part of the WebRTC proof of concept project.

I would like to thank my thesis supervisor Jörg Ott for the valuable feedback and guidance. It was a pleasure to work with him.

Tuomas Erke has instructed my thesis, and I would like to thank him for all the effort and the valuable comments he has given during this work.

I would also like to thank all the people who have been working with me on this subject, and helping me on the way, especially: Jouni Roivas, Mikko Ahola, Mikael Nordman, Ulf Falk, Henrik Taubert, Ville Pelkonen, Sirkku Perttilä, Kari-Pekka Perttula, Juan Osorio Robles, Mia Meinander, Afaque Hussain, Mika Niemi, Jukka Nousiainen, Juha Mäenpää and Miika-Petteri Matikainen.

Finally, I want to express my deepest gratitude to all my friends and family for all the support I have got during my studies.

Espoo, 19.5.2014

Tuukka Koistinen

Contents

Abbreviations

API	Application Programming Interface
AS	Application Server
AVPF	Audio-Visual Profile with Feedback
CNAME	Canonical Name
CPU	Central Processing Unit
CSCF	Call Session Control Function
CSRC	Contributing Source
DNS	Domain Name System
DTLS	Datagram Transport Layer Security
HD	High-definition
HSS	Home Subscriber Server
HTTP	Hypertext Transfer Protocol
ICE	Interactive Connectivity Establishment
IETF	Internet Engineering Task Force
IMS	IP Multimedia Subsystem
JSEP	Javascript Session Establishment Protocol
MC	Media Controller
MKI	Master Key Identifier
MMTelGW	Multimedia Telephony Gateway
MTI	Mandatory To Implement
MTU	Maximum Transmission Unit
NAT	Network Address Translation
PSTN	Public Switched Telephone Network
PT	Payload Type
REST	Representational State Transfer
RR	Receiver Report
RTCP	RTP Control Protocol
RTCWEB	Real-Time Communication in WEB-browsers
RTP	Real-time Transport Protocol
RTT	Round Trip Time
RX	Receiver
SAVPF	Secure Audio-Visual Profile with Feedback
SD	Standard-definition
SDES	Source Description
SDP	Session Description Protocol
SDS	Service Development Studio
SIP	Session Initiation Protocol
SR	Sender Report
SRTP	Secure Real-time Transport Protocol
SSRC	Synchronization Source
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
TLS	Transport Layer Security

TURN	Traversal Using Relays around NAT
TX	Transmitter
UDP	User Datagram Protocol
VoIP	Voice over IP
W3C	World Wide Web Consortium
WCG	Web Communication Gateway
WebRTC	Web Real-Time Communication
XMPP	Extensible Messaging and Presence Protocol

1 Introduction

Real-time communication has become everyday activity in the Internet. Text-based messaging is already an integral part of many social network web pages, and various instant messaging applications exist. Many of those instant messaging applications are also Voice over IP (VoIP) clients, offering audio and video call possibilities over the Internet, and thus bypassing the traditional telephone networks.

In spite of, or perhaps because of the variety of different VoIP clients, web browsers still rule when it comes to the usage of Internet. That said, several techniques already make it possible to implement multimedia telephony services that are accessible from a web browser. However, the drawback of these different techniques is that they are not built in to the browsers, and thus require plug-ins or browser add-ons to work. Also, because of the proprietary nature of the plug-ins, the interoperability between different services is poor.

From all this has emerged a desire for real-time communication standards, that would allow users to utilize audio and video communication services with a standard, off-the-shelf browser, thus avoiding unnecessary plug-ins. A standardized browser interface would also enable web developers to easily create multimedia services on their web pages, that would be natively supported by any standard compliant browser.

All this is now driven by two working groups, the Real-Time Communication in WEB-browsers (RTCWEB) working group at Internet Engineering Task Force (IETF), which focuses on the protocols that enable real-time communication between the endpoints, and the Web Real-Time Communication (WebRTC) working group at the World Wide Web Consortium (W3C), which develops the Application Programming Interface (API) for the web applications to utilize the browser's functionalities and the underlying system resources, such as a web camera and a microphone.

Connecting these web-based multimedia services with legacy networks such as the IP Multimedia Subsystem (IMS), or all the way to Public Switched Telephone Network (PSTN), opens up new possibilities for end users and operators. For example, calls straight from a web browser to a mobile phone and vice versa would be a tempting feature on both user and operator perspective.

To address these possibilities, Ericsson Research recently developed a prototype called a Multimedia Telephony Gateway (MMTelGW). The idea was to create a gateway between WebRTC-based services and the IMS network. As the WebRTC standards were (and still are) evolving, and the browser manufacturers had not yet implemented full support for all the necessary aspects of WebRTC at that time, the prototype was developed with a subset of required features.

The most important feature that was missing from the prototype was data encryption. The standards require that all the media in WebRTC is encrypted with Secure Real-time Transport Protocol (SRTP), using Datagram Transport Layer Security (DTLS) for key negotiation. Since the prototype did not support the encryption, Ericsson Research also created a compliant client, based on the open source Chromium browser. It was modified to omit data encryption, so that it would work with MMTelGW. Support for H.264 video codec was also added, since it is widely used in other systems, like video conferencing services.

1.1 Objectives and Scope

Since the creation of the MMTelGW, the browsers, namely Chrome and Firefox, have evolved to support most of the so far agreed WebRTC specifications. Therefore, in order for the MMTelGW to function with the current browsers, new functionality, such as data encryption, needs to be added to it.

The objective of this thesis is to add the missing implementation of data plane security to MMTelGW. This is achieved by integrating the DTLS-SRTP functionalities to the existing infrastructure. The implementation details of the encryption and other necessary functionalities, such as connectivity establishment, are described.

To evaluate the capacity of the prototype system, the overall performance is measured. The performance is defined by measuring the usage of the central processing unit (CPU), while establishing one or several multimedia calls via the system. We also aim to determine the overall capacity, i.e. how many calls can one instance of MMTelGW handle. Another interesting aspect is the delay, that MMTelGW causes when processing the media packets.

The performance differences between unencrypted and encrypted multimedia calls are analyzed to give a picture on how the encryption affects the load that MMTelGW is capable of handling. It is expected that the encryption increases the load in such a way, that less encrypted calls can be handled than corresponding unencrypted calls. Also, the delays in the encrypted calls are likely higher than in the unencrypted calls, since the encryption requires additional processing.

Since we are dealing with a prototype system, the main objective is to understand how this kind of system works, what needs to be considered, and if it is feasible to implement the system this way. Rather than setting any strict expectations on the capacity, we are purely interested in the number of simultaneous calls the system can handle. In order to test different kind of scenarios, it is enough for MMTelGW to be able to process a few simultaneous calls. More precisely, if ten or more simultaneous video calls can be made, it is more than acceptable.

1.2 Structure

In this chapter the problem domain was introduced, and the objective and scope of the thesis were defined. The second chapter explains the principles behind the key techniques that are required for WebRTC, and describes the situation with the MMTelGW prototype before writing this thesis. In the third chapter the implementation details of the security functionalities are described. The fourth chapter presents the measurement results and analyzes the impacts of the data encryption to the performance of the MMTelGW. In the fifth chapter we discuss the system's applicability as a WebRTC gateway in real-life scenarios. Finally, in the sixth chapter we summarize the results and draw final conclusions.

2 Background

WebRTC standards do not specify the signaling protocols, which are left to the implementations to decide, as described in *WebRTC Overview* draft [1]. Instead, they define the media plane protocols for encrypting and transferring the data, as well as the protocols used for connection establishment. The key techniques here are DTLS-SRTP and Interactive Connectivity Establishment (ICE). In addition to a general overview on WebRTC, this chapter describes the principles of these key techniques. Also the status of the MMTelGW prototype before writing this thesis is explained.

2.1 WebRTC Overview

The WebRTC specifications aim to provide means for peer-to-peer communication between web applications. A web application in this context is a JavaScript application executed most commonly in a web browser, although it can also be executed in any kind of program or environment that implements the required standards to be able to communicate with other WebRTC applications.

The architectural model of WebRTC is designed so that media travels directly from one peer to another via the most direct route possible. An example of a WebRTC trapezoid presenting this architecture is shown in Figure 1. In addition to the browsers and servers, there might be some intermediate devices, such as Network Address Translators (NAT) and firewalls on the way, but WebRTC is designed to try and find a path through all restricting devices, enabling the peer-to-peer media flow.

Although the media travels directly between the users, a centralized server system is still needed as the rendezvous point for the users. After all, the users need to somehow find out where the other user is so that the connection can be made. This is taken care by the signaling system.

Signaling is handled by the servers that provide the WebRTC services. A browser can connect to the server for example using Hypertext Transfer Protocol (HTTP) or WebSockets. The actual signaling protocol is up to the service provider to decide. The servers can then modify or translate the signals as needed. As the signaling protocols at the server level are not standardized, two different service providers can connect their services only if they can agree upon the protocols for inter-server signaling. Commonly used protocols such as Session Initiation Protocol (SIP) or Extensible Messaging and Presence Protocol (XMPP) are proposed as candidates between servers [1].

As mentioned in introduction, two sets of specifications are managed by the two working groups: the JavaScript APIs for the web applications wishing to use the browser's resources, and the standards defining the protocols used between the browsers on the media path.

The WebRTC working group at W3C defines two key specifications for WebRTC: *Media Capture and Streams* working draft defines the JavaScript API that the web application can use to get access to local devices that can generate multimedia

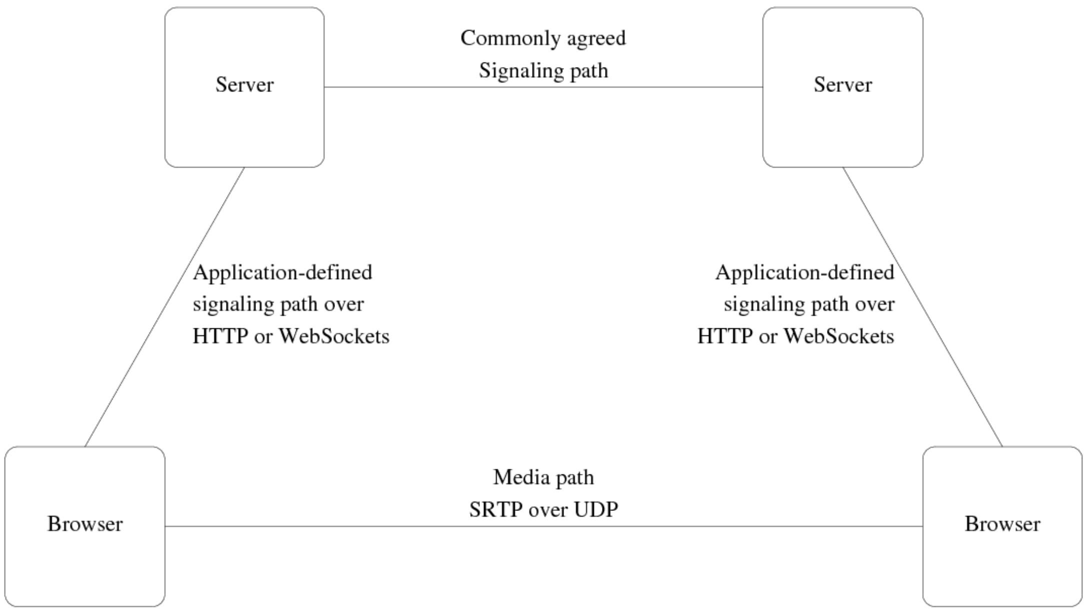

Figure 1: Example of a WebRTC framework overview.

stream data [2], and *WebRTC 1.0: Real-time Communication Between Browsers* working draft defines the JavaScript API that allows the web application to create a connection and send media over the network to another browser or device implementing the appropriate set of real-time protocols, and media to be received from another browser or device [3].

The RTCWEB working group at IETF develops the specifications for the protocols used on the media plane. These include the media transmission protocols, encryption protocols and connection establishment and management protocols. The RTCWEB working group also specifies data channel protocols, which are not discussed in this thesis.

The IETF specifications define two mandatory to implement (MTI) audio codecs for WebRTC usage: G.711 and Opus [4]. Other codecs may also be used, but they will be chosen for the media path only if both endpoints support the same codec and choose to use it as the preferred codec.

Mandatory to implement video codec is not yet agreed upon between the standardization members. Some browser manufacturers are in favor of VP8, being an open source and royalty free codec [5], whereas some prefer H.264 (mostly companies holding patent rights to it) because it is widely used in the Internet [6]. Currently browsers like Firefox and Chrome support only VP8.

However, at the moment H.264 has a slight advance in the race for the MTI video codec. In addition to being the de facto industry standard video codec, Cisco has now released an open source H.264 implementation, from which they will provide binary releases, free of any licensing fees. This would grant any WebRTC application free H.264 support. [7]

2.2 Session Establishment

The actual signaling protocols are left unspecified in the standards, so it is up to the application developers to determine which protocols they decide to use. However, the session establishment, which is carried out on the signaling path, is well defined.

When an end-user wants to use a WebRTC service, it first goes to the web page hosted by the service provider. That web page contains the web application, i.e. the JavaScript application, that will be executed in the end-user's browser.

The JavaScript APIs, which are being specified by W3C, define the interface between the web application and the browser. With these APIs, the service providers, or basically anyone, can create their own WebRTC applications that can be used with any compliant browser.

The web applications operate on so called *RTCPeerConnections*, that represent the peer-to-peer connection in a WebRTC session. The RTCPeerConnection takes care of the session establishment and the media transportation between the web applications [3]. The general operation of the RTCPeerConnection is described in *Javascript Session Establishment Protocol* (JSEP) draft [8].

The session establishment is carried out by exchanging session descriptions that are presented in *Session Description Protocol* (SDP) [9]. These session descriptions contain the connection and media settings each peer is willing to use. The SDP descriptions are exchanged in the fashion described by *An Offer/Answer Model with the Session Description Protocol (SDP)* [10].

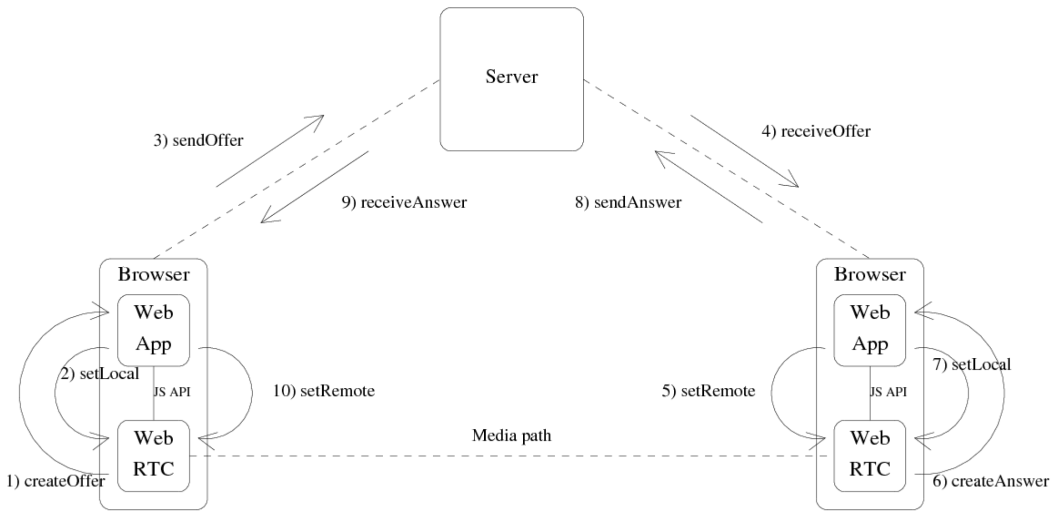

Figure 2: JavaScript Session Establishment

The session establishment procedure is presented in Figure 2. In a normal call scenario, the caller side JavaScript web application (WebApp) first calls a function that causes the browser's native WebRTC module (WebRTC) to create an SDP offer. This offer is given to the web application 1), which has an option to modify the offer if needed. It is yet to be defined what parts of the SDP offer can be modified at this stage, but possible reasons for the web application to modify the browser's generated offer could be to disable some features or discard some codecs that the browser is offering. That offer is then given back to the browser as the local description 2), so that the browser can set up the local configuration according to the SDP. The same offer is then sent to the answer side over the signaling path 3).

When the answer side receives the offer 4), it sets the SDP as the remote description 5). When the call is accepted, the answer side calls a function that causes the browser to create an SDP answer 6), that corresponds to the received offer. This answer can then be set as the local description at the answer side 7), and sent back to the originating side 8). When the caller gets the answer 9), it sets the SDP answer as the remote description 10), and the initial setup is done and the media path can be used.

2.3 Media Transportation

Since WebRTC is all about real-time communication, the media needs to be transported with a protocol suited for real-time data, such as interactive audio or video. WebRTC uses the Real-time Transport Protocol (RTP) to transmit the media between peers. RTP is tightly coupled with its sister protocol RTP Control Protocol (RTCP). In this section we describe these protocols.

2.3.1 Real-time Transport Protocol

RTP is a widely used protocol for streaming media over IP networks, and it is the base of many VoIP systems. It provides end-to-end delivery services, such as payload type identification, sequence numbering, timestamping and delivery monitoring. RTP is defined in RFC 3550 [11].

RTP as such is a protocol framework that is not complete by design. It only defines functions expected to be common across all the applications for which RTP would be appropriate. RTP is intended to be tailored through modifications and/or additions to the headers to suit the needs of a particular application. A complete specification of RTP for said application will require one or more companion documents. Usually this means a profile specification document, which defines a set of payload type codes and their mapping to payload formats.

Each RTP packet consists of a fixed header, followed by the payload data. The RTP header is presented in Figure 3. The meaning of the header fields are briefly explained in the following list:

- Version (V): Version number of RTP. RFC 3550 specifies version two (2) of RTP.

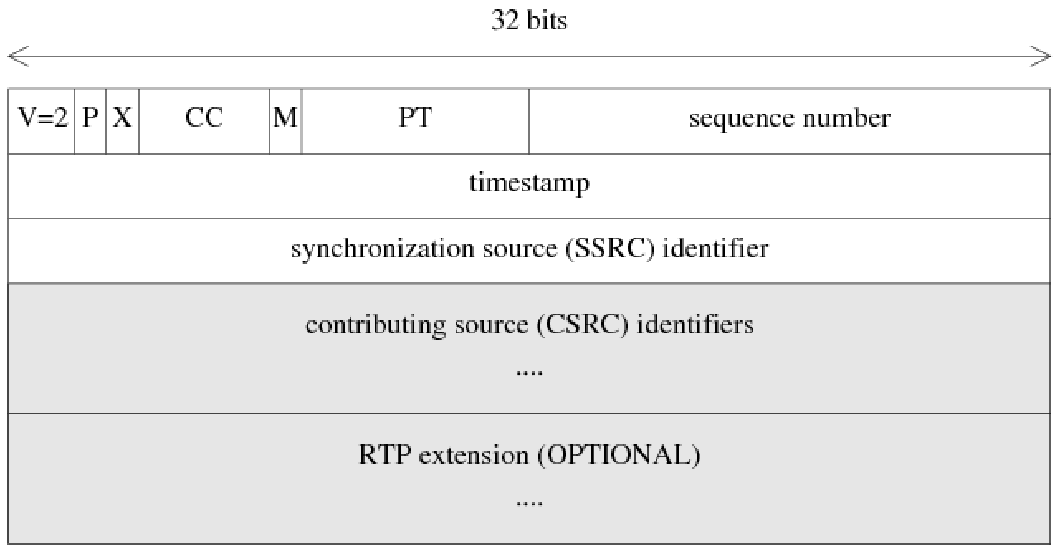
32 bits
V=2
P
X
CC
M
PT
sequence number
timestamp
synchronization source (SSRC) identifier
contributing source (CSRC) identifiers
....
RTP extension (OPTIONAL)
....

Figure 3: RTP header [11]

- Padding (P): The padding bit defines if there is additional padding octets after the payload. The last octet of the padding contains the count of the padding octets, including itself.
- Extension (X): If the extension bit is set, exactly one RTP extension field should follow the fixed header.
- CSRC count (CC): The CSRC count contains the number of CSRC identifiers.
- Marker (M): The interpretation of the marker bit is profile specific. It can be used to indicate significant events such as frame boundaries etc.
- Payload type (PT): Payload type number defines the format of the payload. The mapping between payload type numbers and payload formats is defined in a profile.
- Sequence number: The sequence number increases by one for each RTP packet sent. It is used to detect packet loss and reconstruct packet sequences. The initial value should be randomly selected.
- Timestamp: The timestamp reflects the sampling instant of the first octet in the RTP data packet. The initial value of the timestamp should be random.

- Synchronization source (SSRC): The SSRC field defines the synchronization source. It identifies the source of a stream of RTP packets, for example a microphone or a camera. This identifier should be chosen randomly.
- Contributing source (CSRC): The CSRC list identifies the contributing sources for the payload contained in this packet. The list is inserted only by mixers, which produce combined streams from several sources. The number of identifiers is given by the CC field.
- RTP extension: Optional, application specific field to carry additional data. Can be ignored by other applications.

The first twelve octets of the header are always present in an RTP packet. The CSRC identifiers field is present only when inserted by a mixer, and the RTP extension field can be optionally used by an application if it is needed.

2.3.2 RTP Control Protocol

In addition to RTP, RFC 3550 also defines the RTP Control Protocol (RTCP). It is based on the periodic transmission of control packets to all participants in the session, using the same distribution mechanism as the data packets.

RTCP is intended to provide feedback on the quality of the data distribution. This feedback can be used for example by the sender to control adaptive encoding methods, and also to detect faults in the transmission.

RTCP is also used to carry a persistent transport-level identifier for an RTP source. This identifier is called canonical name (CNAME), and it is used to identify and keep track of the participants in an RTP session. This is needed, because the SSRC identifiers can change during a session in case of a collision or program restart. The CNAME can also be used for associating multiple data streams in a set of related RTP sessions, for purposes like synchronizing audio and video.

Since the number of participants in an RTP session may be large, the RTCP packet sending rate must be controlled to avoid network congestion. By having each participant send its control packets to all the others, each can independently observe the number of participants. This number is used to calculate the rate at which the packets are sent.

RTCP serves as a convenient channel to reach all the participants, since RTP is only sent by media sources. Therefore, RTCP can also be used to convey minimal session control information, for example participant identification to be displayed in the user interface.

There are several types of RTCP packets to be used for the varying purposes:

- Sender reports (SR) are sent by the participants that are active senders of media. Sender reports are used to deliver transmission and reception statistics.
- Receiver reports (RR) are used only for reception statistics from participants that are not active senders.

- Source description items (SDES) are used to identify sources. SDES uses the CNAME item.
- BYE packet indicates end of participation.
- APP stands for application-specific functions. APP packet is intended for experimental use as new applications and new features are developed.

Each RTCP packet begins with a fixed part similar to that of RTP data packets, followed by structured elements that may be of variable length according to the packet type but must end on a 32-bit boundary.

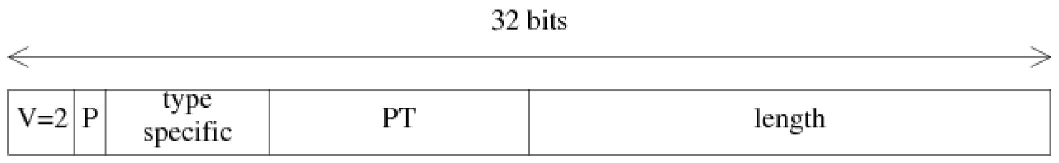

Figure 4: RTCP header [11]

The fixed header part is presented in Figure 4. The version (V) and the padding (P) fields are the same as in RTP header. They are followed by a type specific field, which has different interpretations depending on the RTCP packet type. The packet type (PT) field contains a value that defines the RTCP packet type. The length field contains the length of that RTCP packet in 32-bit words minus one, including the header and any padding.

The alignment requirement and a length field in the fixed part of each packet enable the RTCP packets to be stackable on top of each other, so that many packets can be concatenated to form a compound packet, which can be sent in one lower level packet, for example UDP.

All RTCP packets must be sent in a compound packet of at least two individual packets. The first RTCP packet in the compound packet must always be a SR or RR packet, which makes the RTCP header validation easier. This is true even if no data has been sent or received, in which case an empty RR must be sent, and even if the only other RTCP packet in the compound packet is a BYE. The other mandatory packet that must be included in each compound RTCP packet is an SDES packet containing a CNAME item.

2.4 Media Encryption

RTP transmits the payload in the clear, so systems using plain RTP make the communication vulnerable for eavesdropping by anyone on the way who can access the packets. Therefore, there is a strong desire to encrypt all media traffic in WebRTC. *WebRTC Security Architecture* draft states that “All media channels MUST be secured via SRTP. Media traffic MUST NOT be sent over plain (unencrypted) RTP” [12].

This is further described in *Web Real-Time Communication (WebRTC): Media Transport and Use of RTP* draft, which explains the media transport aspects of the WebRTC framework. It defines the RTP profile for WebRTC to be the Extended Secure RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/SAVPF), which is a combination of the two profiles: the SRTP profile SAVP, and the RTCP-based feedback profile AVPF. Also this draft demands that the full RTP/SAVPF profile needs to be employed to protect all RTP and RTCP packets [13].

SRTP itself is defined in RFC 3711 as a profile of RTP, which provides means for confidentiality, message authentication, and replay protection to the RTP traffic and to the control traffic for RTP, the Real-time Transport Control Protocol (RTCP) [14]. SRTP can be considered as a new layer in the RTP stack, residing between the RTP application and the transport layer. On the sending side, SRTP intercepts RTP packets and then forwards an equivalent SRTP packet to the transport layer. On the receiving side it intercepts SRTP packets and passes an equivalent RTP packet up the stack.

SRTP relies on external key management mechanism for establishing the cryptographic context, i.e. the master key. That master key is then used to derive the session keys, which are used in the cryptographic transformations, such as encryption and message authentication. Several key management mechanisms exist, but in WebRTC it is desired to use only one. The key management is discussed in the next chapter.

The encryption is applied only to the payload of the SRTP packet. The header fields are the same as in RTP packets, and only two additional fields after the payload are defined for SRTP: Master Key Identifier (MKI) and Authentication tag. The authentication is applied to the RTP header and the payload, excluding the SRTP specific fields. The structure of an SRTP packet is presented in Figure 5.

The MKI is an optional field. It identifies the master key, which has been used to derive the session key(s) that authenticate and/or encrypt the particular packet. The authentication tag is a recommended field. It is used to carry message authentication data, and in addition to providing authentication of the RTP header and payload, it indirectly provides replay protection by authenticating the sequence number.

Secure RTCP (SRTCP) provides the same security services to RTCP as SRTP does to RTP. SRTCP message authentication is mandatory and thereby protects the RTCP fields to keep track of membership, provide feedback to RTP senders, or maintain packet sequence counters. [14]

2.5 Encryption Key Management

For SRTP an external key management system needs to be applied to obtain the necessary key information for the encryption and decryption algorithm. For WebRTC,

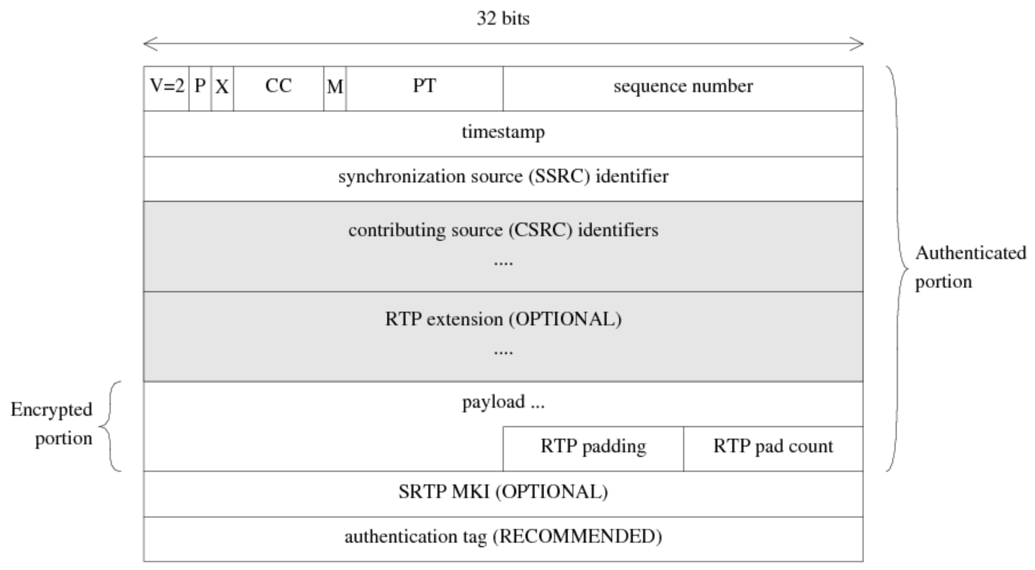

Figure 5: SRTP packet [14]

one mandatory key management system is defined: Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP), also known as DTLS-SRTP [13]. Like the name suggests, it combines the functionality of DTLS and SRTP. DTLS itself is based on Transport Layer Security (TLS) protocol. This section describes these protocols.

2.5.1 TLS

Transport Layer Security (TLS) protocol provides privacy and data integrity between two communicating applications. It is designed to be used over some reliable transport protocol, such as Transmission Control Protocol (TCP). [15]

TLS is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. At the lowest level is the TLS Record Protocol, which ensures the privacy and the reliability of the connection. It takes care of encrypting the data, and checking the message integrity.

The TLS Handshake Protocol allows the server and client to authenticate each other. It is used to negotiate an encryption algorithm and cryptographic keys, that are needed for the data encryption on the TLS Record Protocol layer. TLS Handshake Protocol provides authentication of the peer's identity using asymmetric, or public key, cryptography. The negotiation of the shared secret is both secure and reliable: the negotiation secret is unavailable to eavesdroppers, and the negotiation communication cannot be modified without being detected by the parties of the communication.

TLS is independent of the application protocol that runs on top of it. Once the TLS handshake is complete, the application protocol can transmit its data securely over TLS.

2.5.2 DTLS

Datagram Transport Layer Security (DTLS) protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery, just like TLS does. DTLS-SRTP is based on DTLS 1.0, which in turn is based on TLS 1.1. [16]

Unlike TLS, DTLS provides communication privacy over datagram protocols, such as User Datagram Protocol (UDP). Thus, DTLS preserves the possibility of unreliable and out-of-order delivery of data on application layer.

Since DTLS is very similar to TLS, instead of presenting DTLS as a new protocol, the specification presents it as a series of deltas from TLS 1.1. If not mentioned otherwise, DTLS is the same as TLS.

The DTLS handshake for a DTLS-SRTP session is presented in Figure 6. Note that '*' indicates messages that are not always sent in DTLS. The CertificateRequest, client and server Certificates, and CertificateVerify will be sent in DTLS-SRTP sessions. The DTLS-SRTP protocol is explained in details in the next section.

The ClientHello message contains a list of cipher suites the client wishes to use for the data encryption. The server selects one supported cipher suite from that list, and includes it in the ServerHello message. That cipher suite is the used to encrypt the DTLS data.

In DTLS-SRTP sessions the ClientHello and the ServerHello messages also contain a use_srtp extension, which is used to negotiate the protection profile for the SRTP encryption algorithm. Section 3.3.4 elaborates more on the protection profiles.

In the DTLS negotiation the peers use x.509 certificates to exchange the encryption keys. In the handshake procedure, both the server and the client send their certificates to the other party.

In addition to the certificate exchange on the media path, the peers send certificate fingerprints on the signaling path. A fingerprint is a hash of the certificate, and it binds the DTLS key exchange on the media plane to the signaling plane. [17]

2.5.3 DTLS-SRTP

Datagram Transport Layer Security Extension to Establish Keys for the Secure Real-time Transport Protocol (DTLS-SRTP) is defined in RFC 5764 [18], and as the name suggests, it is an extension to DTLS.

Each DTLS-SRTP session protects one source and destination port pair, generally one RTP or RTCP flow. Each session contains one DTLS association, i.e. DTLS protected connection, and one or two SRTP contexts. Each SRTP context protects all SRTP traffic flowing in one direction on that given source and destination port pair. So for a bidirectional DTLS-SRTP session two SRTP contexts are needed.

To establish the DTLS association the peers perform the DTLS handshake on the given source and destination port pair. The DTLS association can the be used

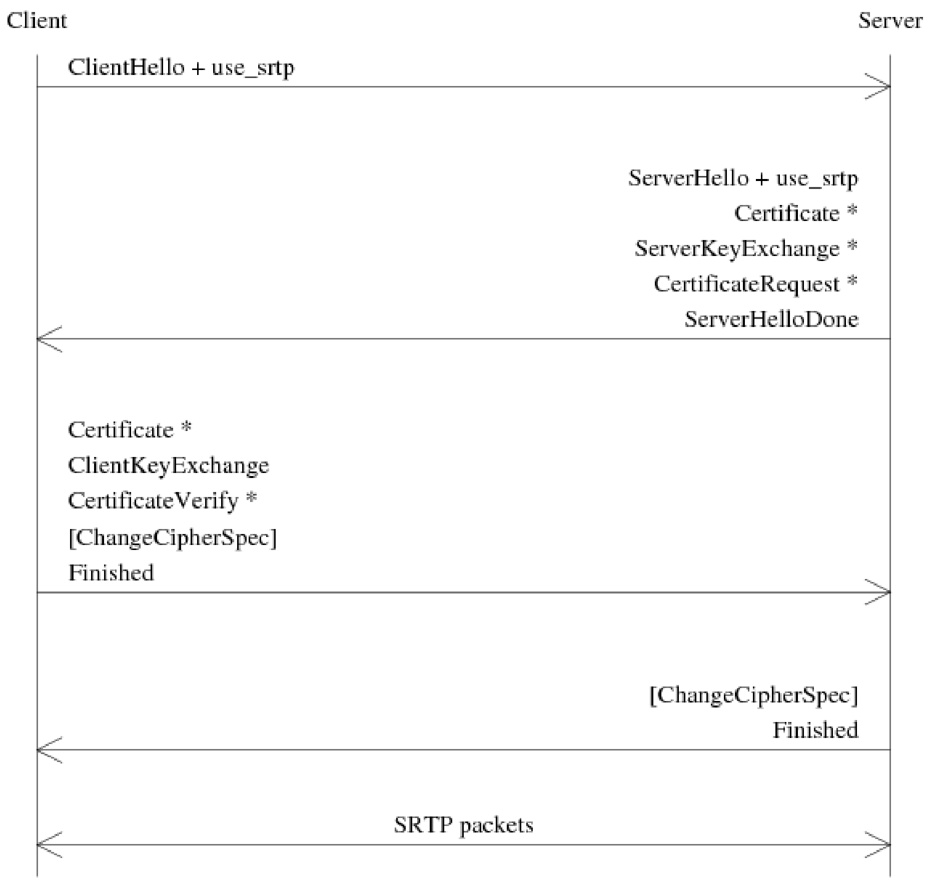

Figure 6: DTLS handshake sequence for DTLS-SRTP session [18]

to generate the keying material for the SRTP stack. The keying material consists of the master keys and master salts for both the client and the server.

The keying material is then given to the SRTP key derivation mechanism, which produces the SRTP (and SRTCP) client and server write keys. The client write keys are used by the client to encrypt and authenticate outgoing packets, and by the server to decrypt and to check the authenticity of incoming packets. Similarly, the server write keys are used by the server to encrypt and authenticate outgoing packets, and by the client to decrypt and to check the authenticity of incoming packets.

Figure 7 presents a bidirectional client side DTLS-SRTP session: it contains the DTLS association, and one SRTP context for incoming data and another SRTP context for outgoing data. Both of the contexts get the keying material for the SRTP stack from the DTLS association. As this is the client side, incoming data uses the

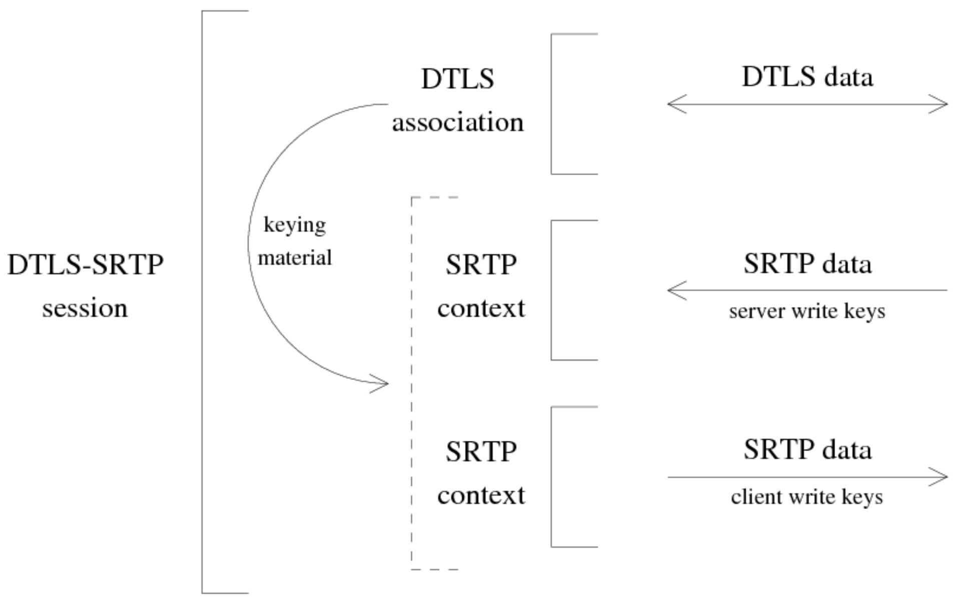

Figure 7: Bidirectional DTLS-SRTP session on client side

server write keys, and outgoing data uses the client write keys. DTLS packets are delivered both ways through the single DTLS association.

Usually, RTCP traffic is sent on a different port than RTP. The usage of different ports requires two DTLS-SRTP sessions, one for RTP and one for RTCP. If the RTP and RTCP traffic are multiplexed to the same port, then a single DTLS-SRTP session can protect both of them.

DTLS operates at the media level on the same host/port quadruple as SRTP, so the DTLS handshake is done directly between the peers. Since the keying material is not passed through the signaling path, the servers used for rendezvous purposes are unaware of the encryption details.

2.6 Interactive Connectivity Establishment

WebRTC aims for peer-to-peer communication, but in the modern Internet peer-to-peer connections are not as simple to establish as one might think. Peers are commonly part of smaller networks, which are secured on the edges by firewalls, and whose address spaces are decoupled from the public Internet by means of NATs. These devices on the network boundaries often block incoming traffic, unless it is part of a connection that was established earlier. To overcome this, WebRTC relies on Interactive Connectivity Establishment (ICE), which is defined in RFC 5245 [19]. In this section we describe the basic concepts of ICE: address candidates, connectivity checks, candidate authentication, different ICE roles and how the connectivity checks are concluded.

2.6.1 ICE Candidates

ICE works by using several address candidates to establish a working connection. In the beginning, ICE agents (the endpoints) gather possible candidates where they can be reached. There are three different types of address candidates: *host candidates*, *server reflexive candidates* and *relayed candidates*. The relationships between different address candidates is presented in Figure 8. The notation X:x means address X and port x.

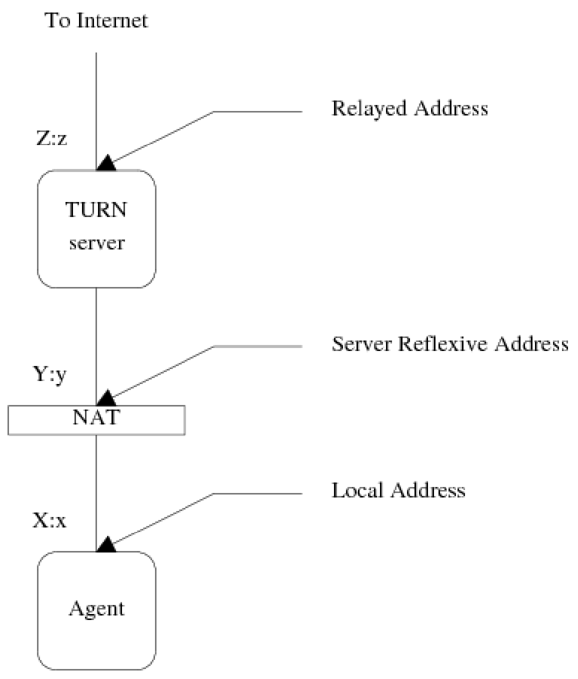

Figure 8: ICE candidate types[19]

Host candidates are derived directly from local network interfaces. In this case the agent's local address X and port x forms the host candidate. ICE also uses Session Traversal Utilities for NAT (STUN) [20] messages to obtain other candidates from appropriate servers. These servers may be simple STUN servers, or they may also implement Traversal Using Relays around NAT (TURN).

When STUN server is used, the agent sends STUN Binding requests to the server, which then replies with a Binding response containing the address where it got the Binding request from. If there is a NAT in between, the address that STUN server returns will be the public address Y:y, that the NAT maps to the agent's local address. That is then called the server reflexive candidate.

When using a TURN server the procedure is largely similar: TURN server also returns server reflexive candidates, but in addition to that, the TURN server is also used to obtain a relayed candidate. When doing that, the TURN server allocates an address Z:z for that media stream. Packets coming to that address from outside are then relayed towards the agent (via the server reflexive address Y:y), and packets coming from the agent are sent onwards from the relayed address. In this thesis TURN is not used.

2.6.2 Connectivity Checks

In the signaling path these candidates are then exchanged inside (SDP) messages. The agents then pair the local and remote candidates, and conduct a series of connectivity checks to see which candidate pairs can be used to establish a connection.

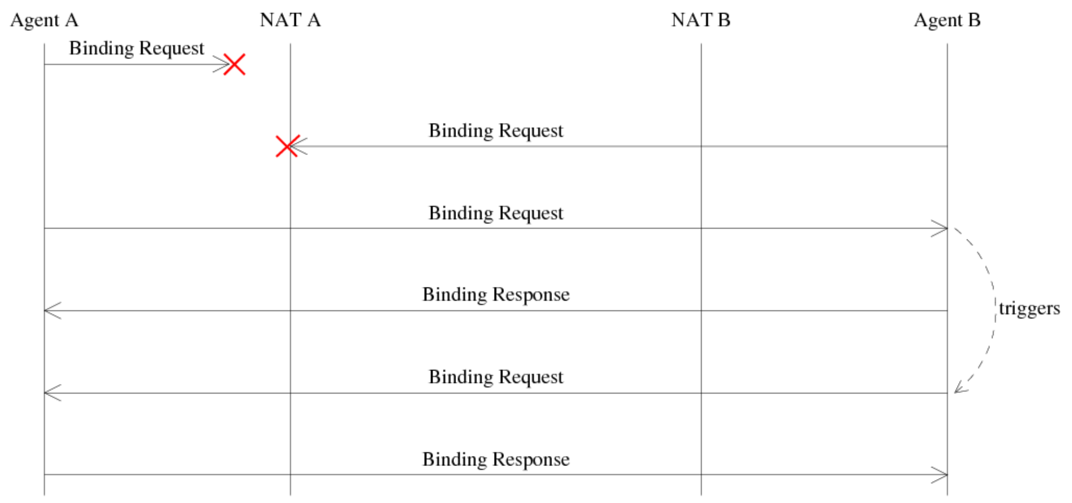

Figure 9: ICE connectivity checks

The Connectivity checks are performed by sending STUN Binding requests to the remote candidates. The procedure is illustrated in Figure 9. The Binding requests may or may not reach their destination due to various reasons. For example, the first Binding request from Agent A might have the Agent B's host candidate as its destination, and because the Agent B's local address is not visible from the outside due to NAT B, the Binding request will never reach its destination. The first Binding request from Agent B might be blocked by a restrictive NAT A, which lets traffic in only if the route has been first opened from within. The third Binding request has a working destination address and the NATs let it through from Agent A to Agent B. When Agent B receives the Binding request, it replies with a Binding response. Receiving a Binding request also means that a Binding request need to be sent to the address from where the Binding request was received, whether that address already was in the candidate list or not. This is called a *triggered check*. If that also goes through and the other agent replies with a Binding response, the four way handshake is done and that candidate pair can be used for the connection.

The connectivity checks are executed in a priority order defined by the agents. This way the agents aim to find the best possible route as fast as possible. However, the highest priority candidates may not always produce the most optimal route. Therefore, the final decision on the selected route can be done according to different practices. Section 2.6.5 discusses the concluding process.

ICE can be implemented either as a full or a lite implementation. The lite implementation is for agents who know that they have a public IP address which is accessible from anywhere. In this case the agent does not gather any ICE candidates, and it only replies to the STUN Binding requests, never sends its own.

2.6.3 Candidate Authentication

ICE has its own authentication mechanism to make sure the candidates are who they claim to be. ICE uses so called short-term credential mechanism to authenticate the STUN connectivity checks between agents [19].

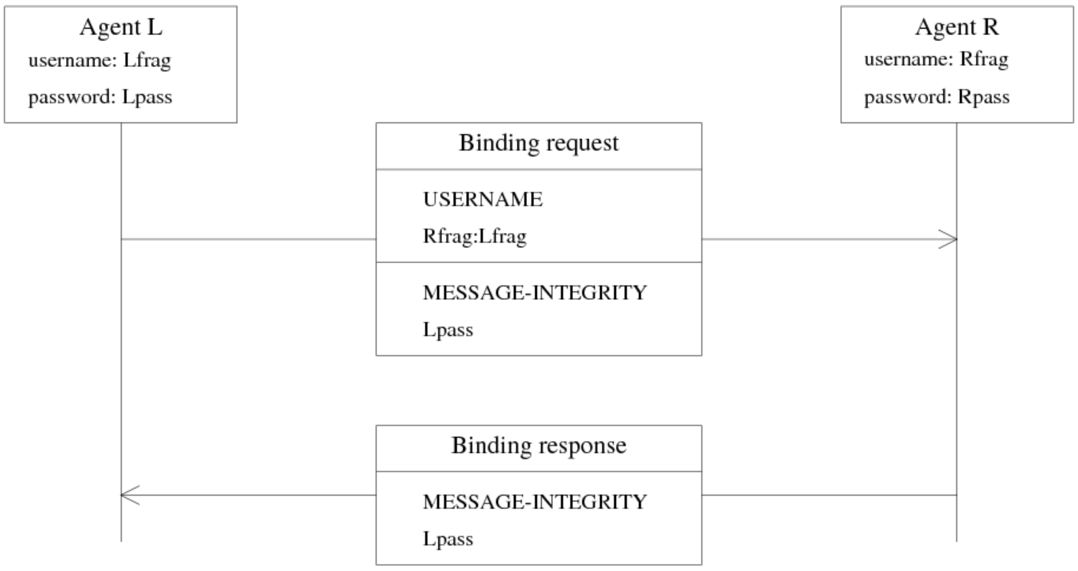

Figure 10: ICE credentials in a connectivity check

Short-term credential mechanism is defined in the STUN specification [20]. It assumes that the endpoints exchange credentials in the form of a username and password using some protocol. In the case of WebRTC and ICE this means that the agents send their ICE username and password in the signaling path inside the SDP messages. These credentials are then used to form the USERNAME and MESSAGE-INTEGRITY attributes in the STUN messages.

In ICE, the Binding requests are always authenticated. The USERNAME attribute is formed by concatenating the username fragment of the receiver with the username of the sender, separated by a colon. The password comes from the sender and is used to calculate the MESSAGE-INTEGRITY attribute. An example of ICE credentials in a connectivity check is presented in Figure 10. All the other STUN attributes are omitted in this example.

The responses utilize the same usernames and passwords as the requests, which means that a response to a Binding request can be generated even if the remote ICE credentials have not yet been received at the time of receiving the Binding request.

2.6.4 Roles

The ICE agents take a role for each session: they can be either controlling or controlled agents. In every session there is one controlling and one controlled agent. The role depends on the types of the agents and which one initiated the session in the following manner:

- **full – full**: the agent who generated the offer will be the controlling agent, and the other one will be the controlled agent.
- **full – lite**: the full agent will always be the controlling one and the lite agent will be the controlled one.
- **lite – lite**: same as with two full agents—the offerer is the controlling and the answerer is the controlled agent.

In case of possible role conflict, the agents try to resolve it using a procedure that involves a tie-breaker value. The tie-breaker is an integer value that both of the agents select randomly. In a conflict situation, the agent with a larger tie-breaker value is set to be the controlling agent.

2.6.5 Concluding ICE

The ICE procedure is concluded when the controlling agent decides that a good enough candidate pair has been found. This can be either the first candidate pair that successfully passes the connectivity check, or it can be based on some other means, such as Round Trip Time (RTT) measurements executed on the successful candidate pairs.

The decision is done by nominating the candidate pair that the controlling agent wishes to use. The nomination is done by setting a *nominated* flag in a Binding request that is sent to the selected candidate. Nomination can be done in two ways: regular nomination and aggressive nomination. In regular nomination, the candidates are first validated with normal Binding requests. When enough validations are done, and a final candidate is selected, another Binding request with the nomination flag is sent again to the selected candidate. In aggressive nomination, the flag is included in every Binding requests. This way, the first candidate that results in a valid candidate pair is automatically selected.

2.7 Packet Multiplexing

The same media path, meaning the IP address and port quadruple between two peers, is used for all the STUN, DTLS and RTP traffic of a single media connection. Therefore, these three packet types need to be multiplexed to the same path on the sender side and then demultiplexed on the receiving side. Multiplexing outgoing packets to one port is trivial, but demultiplexing incoming traffic needs a bit more logic.

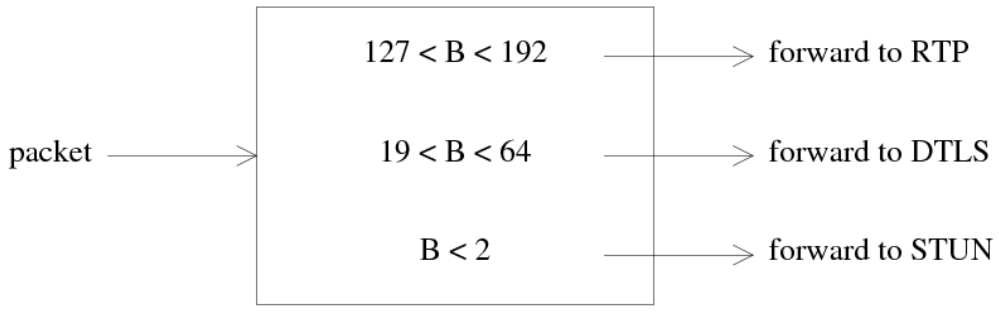

Figure 11: Demultiplexing packets on media plane [18]

In case of WebRTC this is done by examining the first byte B of the packet, as presented in Figure 11. “If the value of this byte is 0 or 1, then the packet is STUN. If the value is in between 128 and 191 (inclusive), then the packet is RTP (or RTCP, if both RTCP and RTP are being multiplexed over the same destination port). If the value is between 20 and 63 (inclusive), the packet is DTLS.” [18]

2.8 Multimedia Telephony Gateway

In this Ericsson prototype system, there are two main components for enabling WebRTC clients such as web browsers to connect to the IMS network: Web Communication Gateway (WCG) and Multimedia Telephony Gateway (MMTelGW). They both reside in between the client and the IMS network, doing the proper transformations in the signaling and the media plane. The system setup is presented in Figure 12.

The client application (WebApp in Figure 12) is a JavaScript application, provided by the service provider (Ericsson in this case). It is downloaded from a web

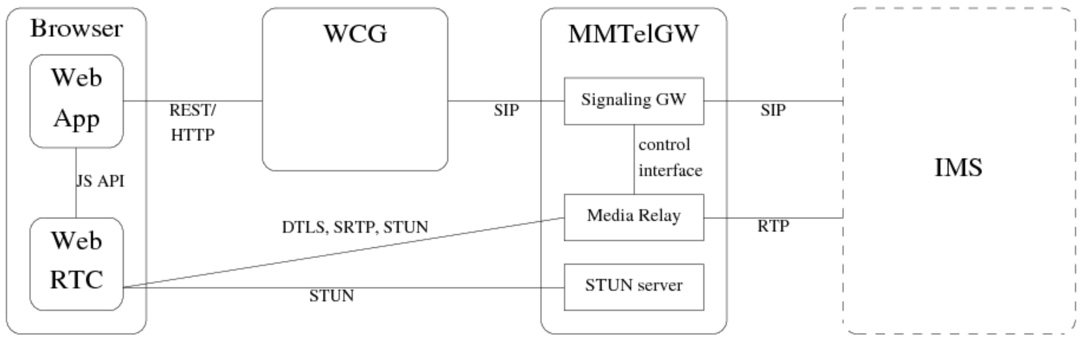

Figure 12: Prototype setup

server, and executed in the browser. The application connects to the WCG through HTTP for the signaling. It also uses the WebRTC JavaScript API to access the browser's WebRTC capabilities, such as the ICE functionality, capture devices and media transportation.

WCG is an actual product developed by Ericsson. It is the connection point for the web clients, and needed because it is unlikely that many web clients would natively communicate using SIP, which is used for signaling on the IMS side. Therefore, WCG provides a Representational State Transfer (REST) interface over HTTP or WebSocket. Upon receiving HTTP requests WCG translates the messages to SIP, and forwards them to the Signaling Gateway in MMTelGW.

MMTelGW is a prototype software, initially developed by Ericsson Research. The purpose of MMTelGW is to translate both the signaling and the media traffic, so that WebRTC clients can communicate with the IMS network. MMtelGW consists of three parts: Signaling Gateway, STUN Server and Media Relay.

The Signaling Gateway takes care of the logic for creating and managing the calls. It handles the SIP signaling and controls the Media Relay over a proprietary control interface.

The STUN Server is a standalone application. Its job is to respond to clients' initial STUN requests, which they make to gather the ICE candidates.

The Media Relay listens to Signaling Gateway and opens media connections between the client and the IMS. It performs media encryption and decryption, because WebRTC clients require SRTP whereas within the IMS media is sent as unencrypted RTP.

When starting this thesis, MMTelGW had several shortcomings such as missing support for DTLS and SRTP. Since the current browsers at that time did not support WebRTC, Ericsson Research forked Chromium browser, and modified it to support H.264 video codec and to pass the data encryption. That browser was then named *Leif*.

2.9 Summary

In the WebRTC architecture the media travels directly between the peers, whereas the signaling goes through a service provider's centralized server system. The WebRTC standards do not specify any protocols for the signaling plane, whereas the media plane protocols are well defined.

The media sessions are established by the JavaScript web applications that are executed in the peers' browsers. An offer/answer model with SDP messages is utilized to negotiate the media parameters for each session.

Media is transported using SRTP, an encrypted version of the Real-Time Transport Protocol, which runs on top of UDP. The encryption details for the SRTP session are negotiated using DTLS, which provides security and data authenticity over UDP.

Today's networks are widely populated by middle boxes such as NATs and firewalls, which may restrict traffic between networks. WebRTC tries to cope with these various network scenarios using Interactive Connectivity Establishment. ICE uses STUN messages to gather address candidates for each endpoint, and then tries to establish connections between the different candidates. The best successful candidate is then chosen to be used for the media traffic.

All three protocols, DTLS, SRTP and STUN run on the same media path. Therefore the WebRTC applications need to be able to separate incoming packets from each other. This can be easily done by inspecting the first byte of the packet.

3 Implementing Security

In this chapter we discuss the implementation details of the necessary features. DTLS-SRTP is a new feature that was added to MMTelGW. Some modifications were needed in the existing ICE implementation to overcome interworking issues between the browsers and our system. Several other issues that arose during development are also addressed in this chapter. Finally, the resulting system is presented, and the call setup procedure is explained.

It is worth noticing that many of the workarounds and features presented in this chapter were implemented to overcome some issues with interworking and incomplete implementations on the browser side. Therefore, some of them might not be needed any more after the specifications are finished and the client implementations mature.

3.1 Need for Security

As described in the scope, the goal of this thesis was to implement necessary features of the specifications, so that the MMTelGW would work with WebRTC compliant commercial browsers, mainly Firefox and Chrome. The main blocker for this is the lack of encryption: the specifications require that all the media in WebRTC sessions are encrypted. Without this encryption, the connection cannot be established towards Firefox or Chrome. Thus, the DTLS-SRTP encryption functionality needs to be implemented to MMTelGW.

Another essential part of the connection setup in WebRTC calls is the interactive connection establishment through restrictive networks, which is performed before the DTLS key negotiation. ICE also has its own security mechanisms to ensure that the offered candidates belong to the session that is about to be initialized. While MMTelGW already has some support for ICE, many parts of the existing ICE functionalities, among other small features, need to be reconsidered because of the different behavior of the browsers

3.2 Software Components

MMTelGW consists of three software components: Media Relay, Signaling Gateway and STUN server, as they are presented in Figure 12.

Media Relay is a Linux software written in C++. It consists of several layers of classes, that represent the sockets and the relays on the lower layers, and the contexts, terminations and streams on the higher layers. It has a control module for the interface towards the Signaling Gateway, and ICE module for the STUN handling. New functionality needs to be added to represent the DTLS and the SRTP protocols, and the existing functionality is modified so that Media Relay could interoperate with the current browsers.

Signaling Gateway is a Java application that takes care of the call handling logic and the signaling modifications that are necessary between the WebRTC clients and the SIP driven IMS network. Signaling Gateway also needs to be modified for some parts, mainly for accepting and handling the necessary SDP attributes.

The STUN server is another C++ software on Linux, running beside the Media Relay. It simply responds to the STUN Binding requests that the clients send in order to find out their server reflexive candidates. The STUN server runs in the default STUN port 3478. That port is set in the JavaScript application, so that the browser can connect to the STUN server and gather the ICE candidates. This part of the software works as it is, and thus does not need any modification.

3.3 Implementing DTLS-SRTP

The MMTelGW needs the DTLS-SRTP protocol to conform to the standards and to be able to interact with the WebRTC enabled browsers. Since the system is a proof of concept and the objective was to get a working solution quickly set up, we wanted to use readily available libraries instead of implementing the whole DTLS-SRTP stack from scratch. In this section we described the used libraries, and how they were integrated to MMTelGW. We also discuss a few changes that were made to the libraries, and the protection profiles that are used in the encryption.

3.3.1 Encryption Libraries

To have a functional security stack, the new libraries need to be integrated to the MMTelGW. Media Relay is the part of the software that handles all the media traffic. Thus it is only logical to integrate the libraries to Media Relay.

For the DTLS, we decided to go with an implementation called *edtls*, created by Ericsson Research. This decision was made since edtls was already used in other projects in the company, and we knew that we could easily get support for it from the original creators.

Edtls is a DTLS library written in C. It depends on OpenSSL, which it uses to construct the DTLS context and to perform the cryptographic functions. Edtls can be used to establish the DTLS connection and to derive the keying material for the SRTP stack.

Edtls can be used with an external SRTP library to offer the whole DTLS-SRTP framework. By default, edtls comes with an SRTP library called *esrtp*, which handles the SRTP stack. Esrtp is also library written in C and developed by Ericsson. We decided to use esrtp for the same reasons as with edtls, and since there already were examples how to combine the two libraries.

Both libraries, edtls and esrtp provide a clear API to handle the structures and the functionalities to achieve the needed cryptographic security.

The edtls library provides a context structure for the DTLS. One context represents one DTLS association. The context takes care of the DTLS state machine of the association at hand. When created, the context structure is initialized with a given x.509 certificate and a callback function, that the context uses to send out DTLS messages.

3.3.2 Software Architecture

To incorporate edtls to Media Relay we created a new class called DTLSContext, which uses the edtls library to provide the needed functionality to the existing software. DTLSContext is a wrapper around the context structure provided by edtls. It contains the information about the role of the DTLS association, and also the SRTP structures related to that DTLS association.

When DTLSContext receives a DTLS packet, the packet is fed to the context, which updates its internal DTLS state machine accordingly. If the context needs to respond to the DTLS message with another message, it uses the sending callback function to do so.

Another class that was created is SRTPContext, which is a wrapper around the context structure provided by esrtp. SRTPContext naturally represents the SRTP contexts within a DTLS association. This means, that a bidirectional DTLS-SRTP session contains one SRTPContext, which in turn contains two esrtp context structures: one for incoming and one for outgoing SRTP traffic. In Figure 7 the dashed line represents the SRTPContext in relation to the rest of the DTLS-SRTP session.

DTLSContext creates and owns the SRTPContexts. If the media stream uses RTCP multiplexing, there are two SRTPContexts: one for SRTP and one for SRTCP. If SRTP and SRTCP are using separate ports, they both use their own DTLS association, and thus reside in their own DTLSContext with a single SRTP-Context.

When the DTLS connection setup is finished, the DTLS context can be used to derive the SRTP keying material. This keying material is then given to the SRTPContexts, which setup the SRTP stack with the given keys. Any incoming or outgoing SRTP or SRTCP packets are then transferred through the corresponding SRTPContexts, which take care of encrypting and decrypting the RTP data.

ICEModule is a module that already existed in Media Relay. The incoming traffic from the media plane is forwarded to the ICEModule, which then takes care of demultiplexing the incoming traffic, as explained in Section 2.7. There is one ICEModule for each media stream, and the ICEModule owns the DTLSContext associated with that media stream.

ICEModule redirects both DTLS and SRTP packets to DTLSContext, which handles the packets as described above. STUN messages are handled in the ICE-Module, which holds the STUN protocol stack. The STUN protocol stack consumes the STUN messages and responds to the Binding requests, which are used for the ICE negotiations in the call setup and for consent freshness during the call.

The structure of the new DTLS-SRTP functionality is presented in Figure 13. The ICEModule already existed in the Media Relay. The ICEModule contains the DTLSContext, which orchestrates the whole DTLS-SRTP functionality. The DTLSContext utilizes the edtls library, whereas the SRTPContext utilizes the esrtp library. SRTPContext is owned by the DTLSContext.

OpenSSL has a possibility for hardware accelerating the crypto algorithms. That works on certain machines that also have support for this kind of acceleration. In

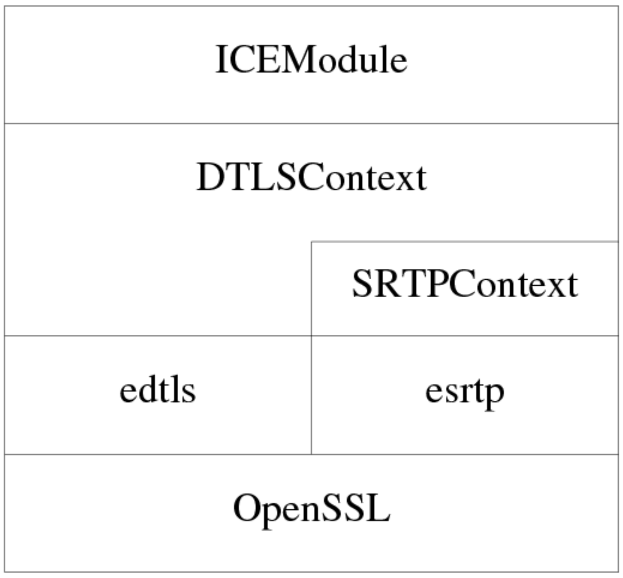

Figure 13: Encryption architecture

order to speed up the encryption and decryption process of the SRTP packets, we enabled the hardware acceleration for SRTP for machines that support it. This required some changes in the esrtp library, which now, in addition to edtls library, also uses OpenSSL functionalities.

3.3.3 Changes in edtls

In DTLS the peers exchange certificates on the media plane and corresponding fingerprints on the signaling plane. Edtls only supported fingerprints that were created with the SHA-1 hash function, whereas Firefox and Chrome both use SHA-256 fingerprints. Therefore, we modified edtls to support also SHA-256 fingerprints.

Originally edtls created new certificate for each DTLS connection. We added an option to use an existing certificate. This was to make troubleshooting easier and to give Media Relay the possibility to use one (possibly configurable) certificate for all the connections.

3.3.4 Protection Profiles

DTLS-SRTP uses protection profiles to determine the characteristics of the SRTP connection. The protection profiles are negotiated within the DTLS handshake, using the use_srtp extension. The client sends a list of supported protection profiles, from which the server selects one to use for the SRTP. If there is no common protection profile, the SRTP connection cannot be established.

Edtls offers two protection profiles for SRTP:

- SRTP_AES128_CM_HMAC_SHA1_80
- SRTP_AES128_CM_HMAC_SHA1_32

These protection profiles are the default profiles defined in RFC 5764, and have the following meaning: SRTP_AES128_CM_HMAC_SHA1_80 uses the AES Counter Mode cipher for encryption with key length of 128 and salt length 112, and the HMAC-SHA1 authentication function with key length 160 and tag length 80. SRTP_AES128_CM_HMAC_SHA1_32 is otherwise the same, but the authentication tag length is 32. All the lengths are in bits. [18]

Two more protection profiles are defined in RFC 5764, but they are NULL cipher profiles, which means that they omit the data encryption and only provide data authentication. Those protection profiles are not supported due to the lack of encryption.

The protection profiles are offered in priority order: the first is the most preferred by the client. In the answer the server selects the highest common protection profile for the session. If no shared protection profile is found, the server should not return the use_srtp extension and the connection falls back to the negotiated DTLS cipher suite. [18]

Media Relay offers the 80 bit long authentication tag profile as the most preferred, and the 32 bit long authentication tag profile as the second option. Firefox has the same priority order. Chrome only offers the 80 bit profile.

3.4 Interoperability

Even though the DTLS-SRTP was the major change needed to interoperate with Firefox and Chrome, there are also other impacts that arise from the differences in the behavior of the browsers. This section describes the features that needed extra attention to get both of the browsers work nicely with Media Relay and each other.

3.4.1 Impacts on ICE

MMTelGW already has some ICE functionality implemented. Basically it has ice-lite capabilities, so it responds to STUN Binding requests but does not gather candidates nor send any Binding requests on its own.

During testing we found out that this is not enough. For some reason the MMTelGW was not able to establish connection towards a terminating Firefox. The connection establishment got stuck to the ICE part, never reaching the DTLS handshake phase.

Turned out that there is a bug in Firefox, which causes it to choose wrong ICE role in a specific case. As described in Section 2.6.4, if one of the ICE agents is ice-lite, the other one has to take the controlling role. In this case, where the Media Relay was acting as an ice-lite agent, the terminating Firefox selected the ICE controlled role, as if it would have been interacting with a full agent.

To overcome this, we decided to implement a bit more functionality to MMTelGW. Instead of being an ice-lite agent, we implemented some of the full agent features, so that MMTelGW could communicate with Firefox.

This meant that instead of only replying to STUN Binding requests, MMTelGW also sends out its own Binding requests to ensure and speed up the connection establishment. MMTelGW also selects the role according to the specification, instead of always being the ice-controlled as ice-lite implementations should.

Since we knew that MMTelGW was going to be used like an ice-lite agent, having a single interface publicly available to all users, we decided that it is not necessary to implement the initial ice candidate gathering. Instead, MMTelGW selects the local host address as the only candidate, just like an ice-lite agent would do. Thus, MMTelGW is not an ice-lite agent, nor is it a complete ice-full agent. This seems to be enough for Firefox, which is now able to establish connection with MMTelGW.

3.4.2 RTCP-MUX

RTP data packets and RTCP control packets are traditionally transferred on different ports: in one RTP session RTP data traffic uses even ports, and the corresponding RTCP control traffic uses the odd port one higher than the RTP port.

Because today's networks are populated with NATs and firewalls, this kind of scenario would cause a lot of overhead, since the NATs should maintain multiple NAT bindings and that is costly. Therefore, it is required for WebRTC implementations to support multiplexing RTP data packets and RTCP control packets on a single port for each RTP session [13]. The feature should however be backwards compatible, so that if rtcp-mux is offered but the answerer does not support it, the session is established without RTCP multiplexing.

RFC 5761 defines a set of means to multiplex RTP and RTCP to the same port. Demultiplexing RTP and RTCP packets is not as straightforward as demultiplexing RTP, STUN and DTLS as described in Section 2.7. There are a few conflicts when distinguishing RTP and RTCP packets. If the clients use dynamic RTP payload types in the range 96-127 or static payload types as proposed in [21] RFC 3551 there should not be any problems. [22]

Both Chrome and Firefox offer rtcp-mux by default. Therefore we decided to implement rtcp-mux also to MMTelGW, since it required only a small effort. Demultiplexing of RTP and RTCP relies on the good behavior regarding the payload types. In terms of signaling it works so that MMTelGW accepts offers with rtcp-mux and acts accordingly, but it does not offer it to others by default. This decision was made because MMTelGW also communicates towards the IMS clients, which may not support rtcp-mux, and may even choke on unrecognized SDP attributes, regardless of the backwards compatibility of the feature.

3.4.3 RTP-BUNDLE

Another way to save resources by reducing the number of transport-layer flows is to multiplex all the RTP media streams in a single RTP session. This means, for example, that in an audio-video call both the audio stream and the video stream is transported to and from the same port. In addition to rtcp-mux, it is required for WebRTC implementations to support this multiplexing [13]. The multiplexing is to be done according to the working draft *Sending Multiple Types of Media in a Single RTP Session* [23].

The signaling for this multiplexing can be done according to the working draft *Multiplexing Negotiation Using Session Description Protocol (SDP) Port Numbers* [24]. In that working draft the SDP Grouping Framework [25] extension called *BUNDLE* is presented.

Chrome offers multiplexing by default, using the BUNDLE format in SDP, whereas Firefox does not offer it. Because the BUNDLE feature is a bit more complex than rtcp-mux and it requires more changes in the signaling plane, we decided to not support BUNDLE in MMTelGW.

The BUNDLE feature is also backwards compatible by the specification, so that implementations that support BUNDLING have to be able to fall back to transport all streams to individual ports. This can be achieved by either just leaving the a=BUNDLE attribute out of the SDP answer, or by stripping the attribute away from the SDP offer that the browser generates, before setting it back as the local description, as described in Section 2.2. MMTelGW omits this parameter in the SDP answers and offers to reject the bundling. We also disabled the BUNDLE in the JavaScript client, so that the BUNDLE offer is not sent in the first place.

3.4.4 Extension Header

When we got the SRTP framework implemented, we noticed that there were interoperability issues between Chrome and Firefox: media sent from Chrome to Firefox was not played back, even though media to other direction worked. Investigating the network traces we noticed, that Chrome uses extension headers in its SRTP packets. For some reason Firefox got confused with the incoming packets and failed to decrypt the data.

We modified the Media Relay so that before sending out any (S)RTP packets it checks if they contain extension header. If they do, the extension header is stripped away before feeding the packet to the SRTP stack. This seemed to fix the issue, and Firefox was satisfied with the SRTP packets and the media plane worked.

The purpose for Chrome to use the extension headers is unknown to us, but since they only matter to Chrome, and removing the extension headers does not affect the media quality, we were confident to do it.

3.4.5 DTLS Role

Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS) [17] describes (among other things) how the endpoints should negotiate whether they take the active or passive role in the DTLS negotiation. This is done using the setup attribute in the SDP message.

The offerer must use the setup attribute value of setup:actpass, which means that it can be either active or passive. The answerer can then choose to use either setup:active or setup:passive. Which ever it chooses to be, the offerer selects the other role. The active endpoint must then start the DTLS negotiation. The answerer is recommended to use the setup:active role, so that the DTLS negotiation can start as soon as possible.

At this point, Firefox takes some shortcuts and assumptions in its implementation. One of these hard coded features is the DTLS role negotiation. Firefox omits this setup attribute from the SDP, and relies on the recommended behavior: the answerer is always the active endpoint, and the offerer is the passive endpoint. This also means that it is always the answerer who initiates the DTLS handshake.

Chrome also acts according to the recommended behavior, but it also includes the setup attribute in its SDP messages.

This had to be taken into account in the Media Relay. In case of missing setup attribute, the default behavior was assumed. Even though Firefox omitted the setup attribute, Media Relay would always respond with the correct setup attribute in the SDP.

3.5 Complete System

The current system provides all the necessary features to establish WebRTC calls with up-to-date browsers, including media plane security and interactive connectivity establishment. This section describes the final system layout after the modifications, and how the different parts work together when setting up a WebRTC call.

3.5.1 Final System Layout

Early in the beginning of our work, the responsibility of the Signaling Gateway development was moved to the people developing WCG. The name of the Signaling Gateway was changed to Media Controller (MC), and the software was coupled with WCG, since both of them operate on the signaling plane. Media Relay stayed as a standalone component, and the MMTelGW as a logical entity was deprecated. The current system setup is presented in Figure 14.

This setup shows one entry point between a browser and the IMS network. In this setup, there are two possible call scenarios: the call is initiated either from the browser side going towards the IMS, or the call comes from the IMS and terminates to the browser. It is irrelevant to the Media Relay and the WCG what is on the other side of the IMS network—it might be another browser connected through another WCG, a VoIP client or something else.

3.5.2 Browser Initiating Call

Putting it all together, a typical successful browser initiated call setup is presented in Figure 15. Before the call setup, the user has registered to the IMS core through the WCG. The call setup begins with the browser sending the initial SDP offer via the HTTP path to WCG. WCG then instructs Media Controller to signal Media

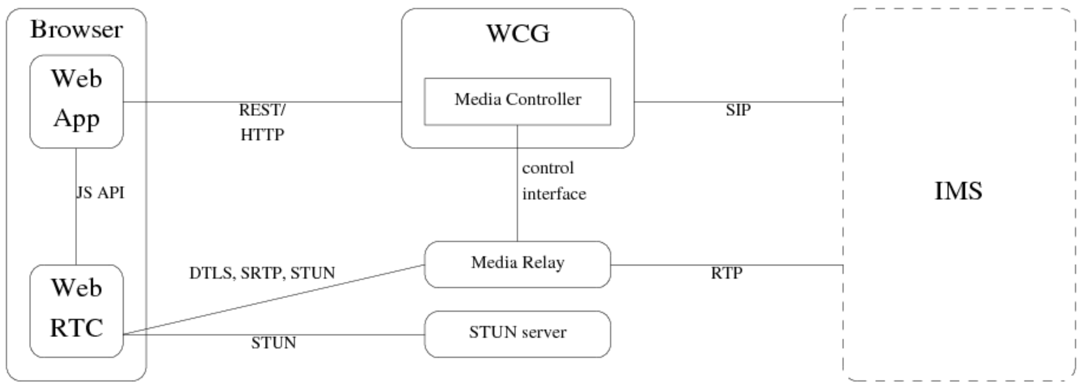

Figure 14: New system setup

Relay to create terminations towards the browser and the IMS. Once this is done, WCG forwards the SDP offer in SIP to the other client behind the IMS.

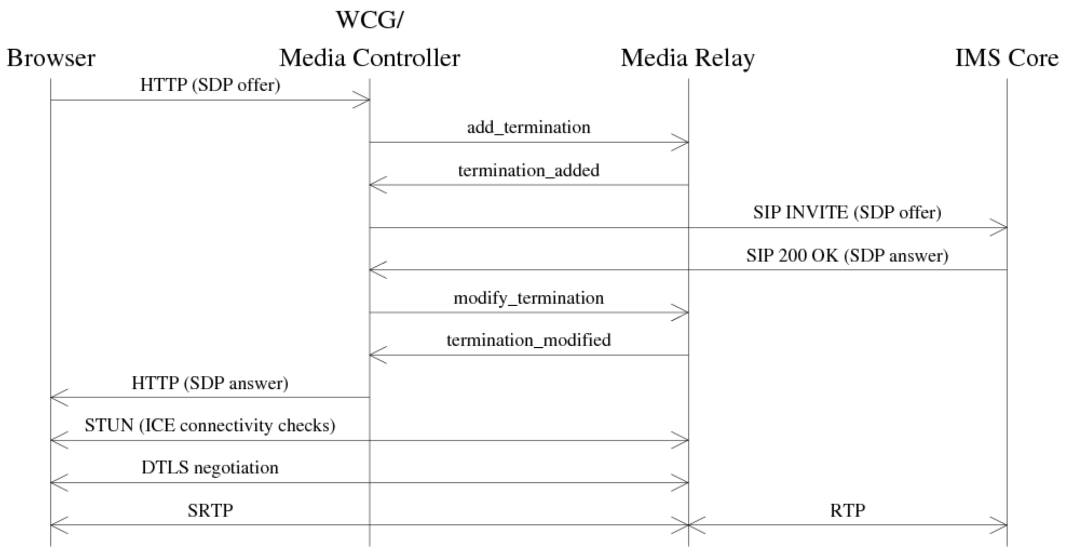

Figure 15: Client originating call

When the SDP answer comes back from the other client, the Media Controller modifies the terminations in Media Relay according to the received SDP answer, which is then forwarded to the originating browser. That concludes the signaling part. Next, the browser and the Media Relay start the ICE connectivity checks to determine the working media path between the two. Once the ICE has concluded, the browser and the Media Relay can start the DTLS negotiation on the specified media path. After the DTLS negotiation finishes, the media can start flowing; as SRTP between the browser and the Media Relay, and as RTP between the Media Relay and the IMS.

3.5.3 Browser Terminating Call

In a call that comes from the IMS side towards the browser (for example the other end of the previous call setup), the setup steps are the same, just the ordering differs. Figure 16 shows how this happens.

The SDP offer comes in a SIP INVITE to the WCG, which causes the Media Controller to create the terminations in the Media Relay. The SDP offer is then sent to the browser, which answers according to its capabilities. Once the terminations are modified, the SDP answer is sent back towards the IMS. Now the ICE connectivity checks can be started, and after finishing that, the DTLS negotiation is executed. The media can start flowing as soon as the DTLS negotiation finishes and the media path is open.

If you match together the two setup scenarios presented here and in Section 3.5.2, the SIP INVITE towards the IMS in Figure 15 is the same as the SIP INVITE that comes from the IMS side in Figure 16. Similarly the SIP 200 OK messages map to one another, and the media flows as RTP between the two Media Relays.

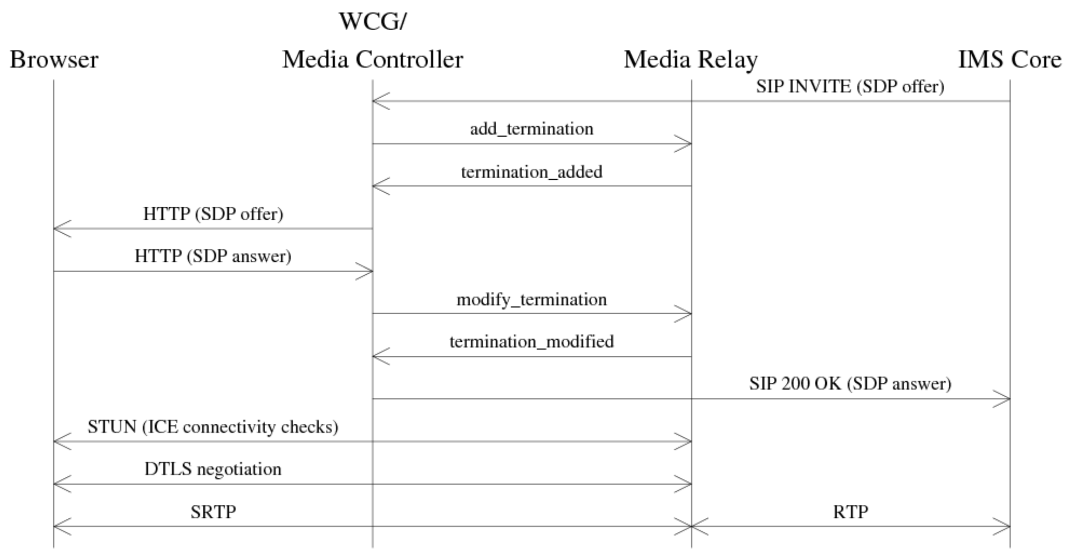

Figure 16: Client terminating call

3.6 Summary

Data encryption is mandatory part of WebRTC communication. The media has to be encrypted using SRTP, and the key negotiation for the SRTP has to be done using DTLS.

This was addressed in the MMTelGW by implementing support for DTLS-SRTP. Media Relay uses two external libraries, both developed by Ericsson: edtls and esrtp. Two main classes, DTLSContext and SRTPContext were implemented to wrap these libraries and to integrate them to the existing code base. Both of the libraries depend on OpenSSL.

Since the standardization is still underway, the browsers are still varying in their behavior. Many workarounds were introduced to achieve interoperability between different browsers, impacting the DTLS-SRTP and ICE functionalities.

MMTelGW as a logical node was deprecated, when Signaling Gateway changed its name to Media Controller and became part of WCG, leaving Media Relay as the single media plane node.

From Media Relay's point of view, every WebRTC call has a direction that characterizes the call: it either comes from a browser and goes towards the IMS, or vice versa. It does not matter whether the other endpoint on the other side of IMS is another WebRTC browser or an IMS client.

4 Measurements and Evaluation

To evaluate the performance of the Media Relay, we conduct a series of measurements with the prototype system. In this chapter we discuss different types of scenarios where WebRTC will probably be used. These typical WebRTC sessions can give characteristics for the overall performance of the Media Relay. Then we describe the prototyping environment that was used for the measurement. After presenting the measurement result we continue to analyzing the results to get a good picture of the Media Relay performance characteristics.

4.1 WebRTC Scenarios

WebRTC is the standard of the future for multimedia communication, and it can be used for varying types of real-time multimedia sessions. Typical multimedia sessions contain different combinations of video, audio and data streams. Naturally video requires more bandwidth than audio, and the bitrate of data streams can vary greatly depending on the nature of the data: short chat messages will most likely generate less traffic than some continuous application data for example in a multiplayer game. In this thesis we focus on audio calls, and video calls which also contain audio.

For calls containing a video stream, the resolution of the video affects greatly the required bandwidth. High-definition (HD) video's bitrate can be measured even in megabits per second, whereas audio streams reach from a few to a few hundred kilobits per second, depending on the used encoding method.

The Media Relay was created as a proof of concept to test the interconnectivity between WebRTC endpoints and the IMS network. Since the Media Relay is not a real product, the performance was not the key element in the development. However, to have some kind of picture of the capabilities of the Media Relay and the bandwidth requirements in typical WebRTC scenarios, some measurements are needed.

In the measurements we are interested in the processing performance metrics of Media Relay. Since the system was first built without data encryption, it is interesting to know how much overhead the encryption adds. Of course this is a rather theoretical point of view, since WebRTC as such cannot be used without encryption due to the mandatory requirements.

Much more concrete value for the measurements comes from the overall performance of the Media Relay: how many calls can one instance of Media Relay handle? What is the processing load and the throughput for different types of calls, e.g. audio calls and video calls?

4.2 Prototyping Environment

To measure the performance of the Media Relay we setup a simple system consisting of one WCG, one Media Relay, the IMS network and several clients. Instead of a real IMS network we use a software-based simulated IMS core, called the Service Development Studio (SDS).

The IMS network as such consists of many parts: the Application Server (AS), the Call Session Control Function (CSCF), the Home Subscriber Server (HSS), and the Domain Name System server (DNS). These components communicate with each other to handle incoming requests. SDS is a development environment based on the Eclipse platform, which is used to simulate the IMS network. It is also developed by Ericsson. In the figures the IMS network is presented as a cloud, which may or may not handle the traffic. The actual functionalities of the IMS network are not relevant for this thesis.

In this section we discuss different network scenarios where Media Relay could be used. Then we describe the test devices that were used in the measurements.

4.2.1 Network Scenarios

Different network setups affect the expected load on the Media Relay. For Every Media Controller there can be N number of Media Relays. Media Controller assigns the call legs to the Media Relays according to some load balancing rules. Media Relays are only needed for WebRTC calls coming from or going to browsers, whereas IMS clients connect straight to the IMS network. Different call scenarios are explained with the help of the following figures.

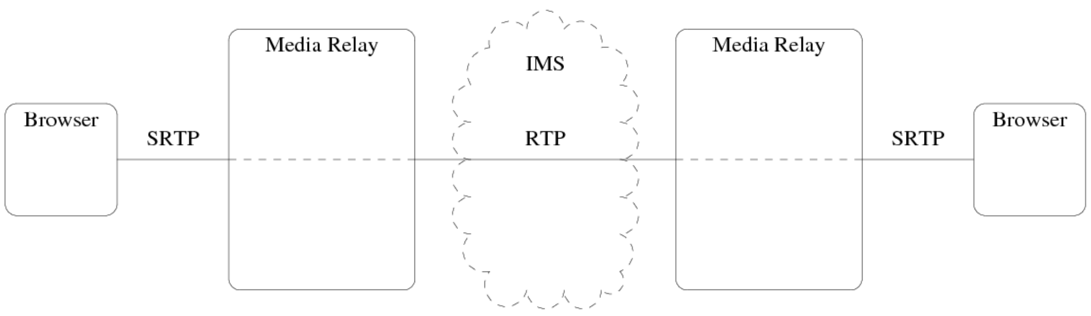

Figure 17: WebRTC call with two Media Relays

If there are several Media Relays in the network, the Media Controller can assign the call legs to any of the Media Relays it controls. In this scenario the clients usually connect to different Media Relays, and the RTP traffic “inside” the network is transmitted between the two Media Relays, as presented in Figure 17.

If only one Media Relay is used, the WebRTC call between two browsers is handled twice in the Media Relay, as presented in Figure 18: the data from one browser is transformed from SRTP to RTP on the way towards the network, but then looped back to the same Media Relay before encoding and sending it to the other browser. This kind of setup doubles the traffic in one Media Relay per call, compared to a situation where both browsers would be connected to separate Media Relays.

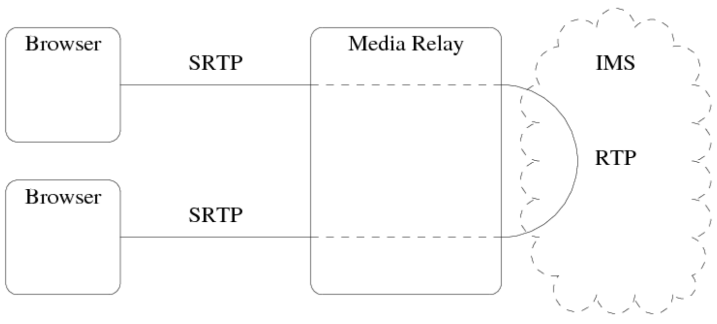

Figure 18: WebRTC call with one Media Relay

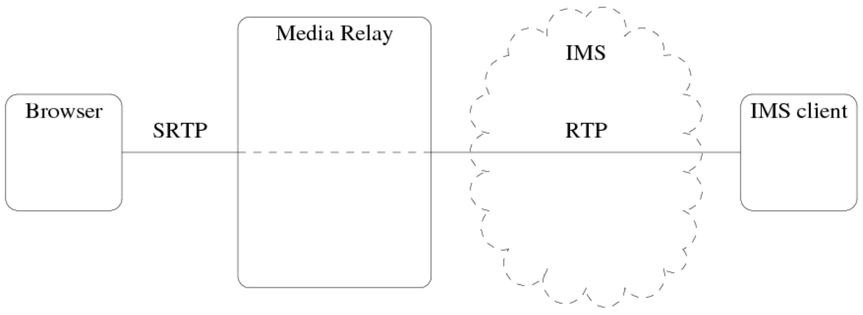

Figure 19: WebRTC call between a browser and an IMS client

When a browser makes a WebRTC call to an IMS client, or vice versa, the browser side behaves exactly as in the browser to browser call: the Media Relay does the encryption and decryption between the browser and the IMS. On the IMS client side, the media goes through the IMS network as plain RTP and is terminated as such in the IMS client, assuming it does not require secure RTP. This is presented in Figure 19.

In our prototyping environment we use the setup with one Media Relay, as presented in Figure 18. This kind of scenario is easy to setup since it only requires one machine for the single Media Relay. It is also feasible for load testing, since single call generates double the amount of traffic—less calls are needed to reach the capacity limit of the Media Relay.

The measurements were carried out in the machine running the Media Relay and the WCG. The next section describes the characteristics of the used machines.

4.2.2 Test Device Description

Both the WCG and the Media Relay are Linux applications. Even though it is possible to run them on separate machines, in our tests they were run on the same machine. The details of the Intel machine running the Media Relay and the WCG on Ubuntu 13.10 are presented in Table 2. The Media Relay machine had the option for OpenSSL hardware acceleration, which was used in all the test calls. All the measurements were conducted on this machine.

Table 2: Media Relay machine specification

	Media Relay PC
Processor	Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
Memory	8 192 MB DDR3 SDRAM 1 600 MHz
Operating System	Ubuntu 13.10 64 bit
Linux Kernel	3.11.0-15-generic

Even though all the measurements are done on the Media Relay machine, the media is generated and transmitted by the client machine. This may be relatively heavy operation, especially in case of HD video. Therefore, the details of the client machine are presented in Table 3. All the single call tests were run on this machine, and the multiple call cases were run using this and other more or less similar client machines.

Table 3: Client machine specification

	Client PC
Processor	Intel(R) Core(TM) i5-2540M CPU @ 2.60GHz
Memory	4 096 MB DDR3 SDRAM 1 600 MHz
Operating System	Windows 7 64 bit

The simulated IMS core was run on a separate machine on top of 64 bit Windows 7. The specifications of the machine are identical to the Media Relay PC.

4.3 Measurement Results

To get a grasp of the order of magnitude where the Media Relay's performance lies, we ran a set of tests with different traffic profiles in our prototyping environment that was presented in Section 4.2. In this section we first describe the test setup. Then we discuss the different traffic profiles that were used in the measurements. Then we present the measurement results, by first analyzing the payload sizes on the different media streams. After that, we examine the throughput of all the call types. Then we focus on the Media Relay's performance, in terms of CPU load and packet delay. Finally, we present the results from the load testing.

4.3.1 Test Setup

Two sets of tests were run to measure the Media Relay's performance under single calls and multiple calls. *Tcpdump* was used to capture all the traffic. From the tcpdump output file we got the size of each RTP packet's payload, and also the delay between receiving and sending out each RTP packet. The RTP packets were identified by their sequence number.

CPU performance was recorded with Linux utility *top*. The CPU load samples of the Media Relay process and the WCG process were measured with a two second time window. The overall traffic sent and received by the machine running the Media Relay and the WCG was measured with another Linux utility called *sar*.

The test setup is presented in Figure 20. Since the prototype system does not have any automatic feature test framework, the test calls need to be done manually. This means that while the Media Relay and the WCG runs on one machine (Gateway Machine), and the simulated IMS on another (IMS Machine), the test calls are initiated from a separate machine (Client Machine). The calls are naturally made between two different users, but in practice both the users are on the same machine using just different browser windows. This was done to simplify the testing, and also because from the Media Relay's point of view it does not matter whether the two clients are on the same machine or two different machines.

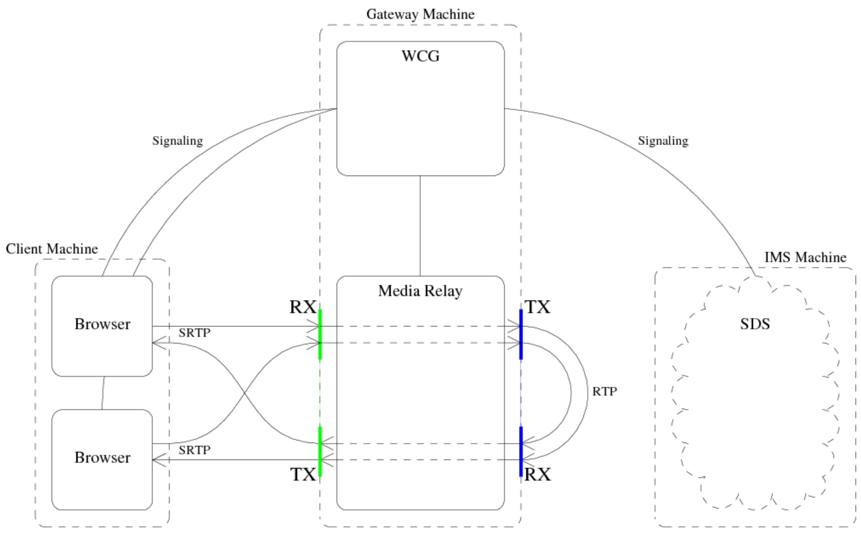

Figure 20: Test setup

The green line represents the public network interface in the Gateway Machine, and the blue line is the local interface. The interfaces are divided into receiver (RX) and transmitter (TX) parts, since they are measured separately.

The single call measurements were conducted so that we measured the above-mentioned statistics over a duration of five minutes from each call. The results of the single call measurements are presented in Sections 4.3.3, 4.3.4 and 4.3.5.

Multiple call scenarios were tested with ten simultaneous calls. The calls were done consecutively, so that each call was done on top of the previous one, and the statistics were measured during one minute after each call setup. The load testing results are presented in Section 4.3.6.

All of the single call tests were made using a Chrome browser, version 33.0.1750.154m as the client, and the multiple call tests were made with the same version, or newer. The client application in use is a simple JavaScript code tailored for basic video calls.

To offer a bit more than a mere average for the measurements, most of the data is presented using a boxplot. They are useful especially for comparing distributions between several groups of data. In the boxplots, the middle line corresponds to the median, the hinges represent the first and the third quartile of the data, and the whiskers extend to the most extreme data point which is no more than 1.5 times the length of the box away from the box. Outliers are included in the figures and plotted as small dots. The boxplot data are also presented in the tables in Appendix A.

4.3.2 Traffic Profiles

In the tests we used five different traffic profiles, which are presented in Table 4. Two unencrypted scenarios include an audio call (A-RTP) and a low resolution video call (SD-RTP). Three encrypted scenarios consists of an audio call (A-SRTP) and two video calls, standard-definition (SD-SRTP) and high-definition (HD-SRTP).

Table 4: Traffic profiles

Name	**Encryption**	**Audio format**	**Video format**	**Video size**	**Browser**
A-RTP	no	PCMU	-	-	Leif
SD-RTP	no	PCMU	H.264	800x600	Leif
A-SRTP	yes	Opus	-	-	Chrome
SD-SRTP	yes	Opus	VP8	800x600	Chrome
HD-SRTP	yes	Opus	VP8	1 280x700	Chrome

To measure the impact of the SRTP encryption and decryption in the Media Relay, we compared the unencrypted and encrypted media sessions. Since Chrome and Firefox transmit the media in SRTP by default, as it should be, we used the Ericsson modified Chromium browser *Leif* to test the performance of plain RTP traffic.

Unfortunately the supported codecs did not match between Chrome and Leif: Chrome uses VP8 for the video and Opus for the audio, whereas Leif uses H.264 for the video and PCMU for the audio. Therefore the results between RTP and

SRTP calls are not directly comparable, but nevertheless provide an indication of the performance characteristics. Also, Leif is based on an older version of Chrome, and therefore the JavaScript application for Leif does not support HD video, which Chrome does support.

4.3.3 Payload Size Analysis

Figure 21 presents the audio packet payload sizes in the different call scenarios. The graph contains also the video calls, but presents only the payload sizes of the audio packets in them.

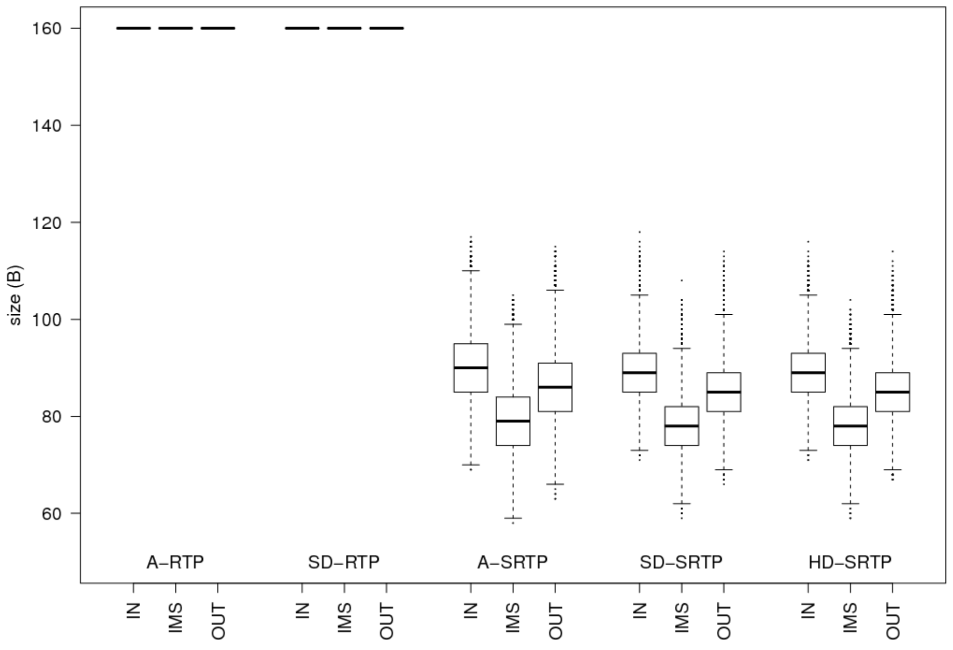

Figure 21: Payload size of audio packets

The different call scenarios are divided into three parts: *IN*, *IMS* and *OUT*. The IN part shows the packet payload size when the packet comes from the client towards the Media Relay. The IMS part shows the payload size on the IMS side of the network, which in our test scenario means when sending the packet from the Media Relay to itself. In a multi Media Relay scenario this would happen between two different Media Relays. The OUT part shows the payload size when Media Relay sends the packet towards the client.

As can be seen, the RTP calls contain audio packets with 160 byte payload, whereas the SRTP calls use mostly less than 100 bytes per packet. The RTP calls use PCMU encoding for the audio, and the payload size for PCMU is constant—not even any outliers are detected. SRTP calls on the other hand use Opus audio codec, which clearly has more variance in its payload sizes, and outliers both below and above the whisker range.

In the RTP calls the payload size is the same in all the three parts. This is clear, since in the RTP calls, Media Relay only relays the traffic without any payload tampering. In the SRTP calls, however, there is clear a difference between the three parts. The IMS part is naturally lower than the two other parts. This is because the Media Relay terminates the SRTP connection, and therefore the IMS side traffic is plain RTP. SRTP packets are generally four to ten bytes larger, depending on the authentication tag size (32 bit tag versus 80 bit tag, as described in Section 3.3.4).

The difference between IN and OUT parts are likely caused by the extension header removal. As described in Section 3.4.4, Chrome uses extension headers, size of eight bytes, in its SRTP packets. These extension headers are, however, removed in the Media Relay, which results the outgoing packets being smaller than the incoming packets. Even though the extension header is technically not part of the payload, tcpdump's RTP interpretation includes the extension header to the payload byte count.

When looking at the SRTP calls, the audio payload sizes seems to be the same in the SD and HD video calls. Also the audio only call matches the payload sizes of the video call, which seems to indicate that there is no difference in the audio encoding between the different call scenarios. This seems to hold true also for the RTP calls.

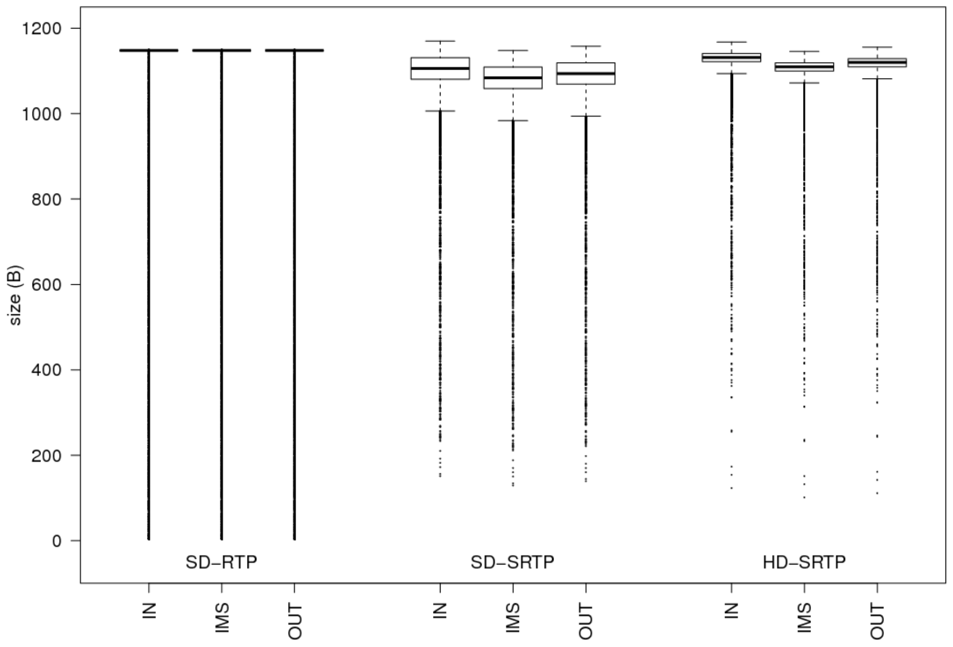

Figure 22: Payload size of video packets

Figure 22 shows the payload sizes for the video packets. The data is in line with the audio payload sizes: there is no difference between the IN, IMS and OUT parts in the RTP call, but in the SRTP calls the packets on the IMS side are smaller than incoming and outgoing packets, and the outgoing packets are smaller than the incoming packets due to the missing extension headers.

The H.264 video packets in the RTP call are also constant in size between the different parts, whereas the VP8 video packets in the SRTP calls have some variance. This is the same behavior that is seen in the audio packets: IMS packets are RTP whereas in and out packets are SRTP, and outgoing packets do not have the extension header what incoming packets may have. The HD video packets are slightly larger than the SD video packets, but they also have a little less variance. The maximum sizes in the SRTP calls seem to be on the same level.

All of the video packets, both H.264 in RTP calls and VP8 in SRTP calls have a wide range of outliers below the lower whisker. Even though most of the video packets' payload sizes sit above thousand bytes, the outliers payload size range span even close to zero bytes. This probably happens because larger video packets need to be split in to several RTP packets. So, while most of the packets are close to the maximum payload size, probably limited by the maximum transmission unit (MTU), some of the packets carry the remaining of the video packet fragments, and therefore spread below the more common packet sizes.

4.3.4 Throughput

To measure the traffic load on Media Relay, we monitored the network interfaces with a Linux tool called *sar*. It records the RX and TX data statistics, and reports them by the interface. The results do contain all the recorded network traffic in those interfaces, so the RX and TX statistics contain a little overhead to the actual WebRTC media.

Figure 23 presents the traffic data in packets per second. The different call types are further divided into three parts: *RX*, *TX* and *IMS*. RX denotes the received traffic on the public interface, and TX the transmitted traffic on the public interface. Since the IMS side traffic was sent from the Media Relay to itself, it was done using the local interface. Also, since the transmitted data equals the received data when transferred within the loopback interface, the IMS side traffic is only presented once in the graph. This setup can be seen in Figure 20: the RX and TX statistics correspond to the RX and TX of the green interface, and the IMS statistics come from the RTP path between the TX and RX of the blue interface.

The graph shows clearly the differences between the call types: audio calls transfer around 100 packets per second per direction, whereas SD video calls get close to 300 packets per second in each direction. In HD video calls the average packet rate per second sits around 500 packets for every direction.

The IMS side packet rate is lower than the packet rate in the public interface in all the call scenarios, since there is less extra traffic in the loopback interface. This extra traffic in the public interface consists mostly of STUN messages, which are periodically sent during the call to check that the connection stays alive. The

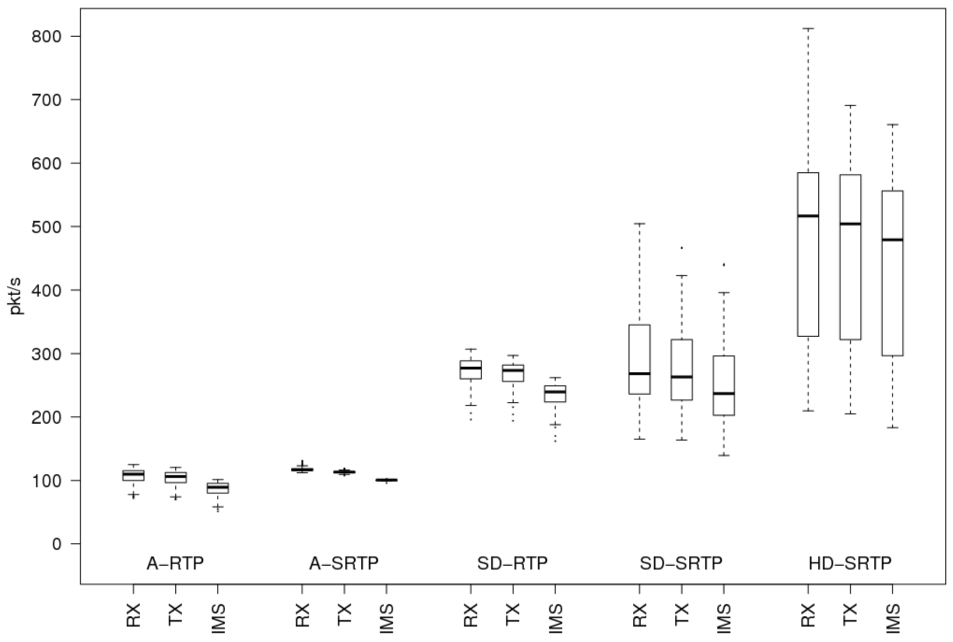

Figure 23: Throughput in packets per second

STUN messages are only needed between the Media Relay and the WebRTC clients, so the IMS side is free of all the keepalive messages. In all the call scenarios the received traffic has a slightly higher rate than the transmitted traffic, but this might be caused by the other normal traffic in the network.

In addition to packets per second, *sar* also reports the traffic in kilobytes per second. Figure 24 presents the throughput in kilobytes per second, and the result is very much in line with the packet rate graph.

When taking into account that the IMS traffic in the graph is doubled when calculating the total sent and received traffic on the local interface, and that the median rate of the HD video call on IMS side is 426.07 kB/s, we can say that the total amount of traffic one HD video call generates in the Media Relay, in this kind of network setup, is on average around 1 700 kilobytes per second.

4.3.5 Media Relay Performance

As the main objective of this thesis the performance impacts of the WebRTC calls to Media Relay are evaluated. To achieve this, we measure the CPU load during the different call scenarios, as well as the delay that Media Relay causes to the media transmission.

The average CPU loads of the Media Relay process, generated by the handling of the RTP and SRTP calls are presented in Figure 25. The order of the call scenarios

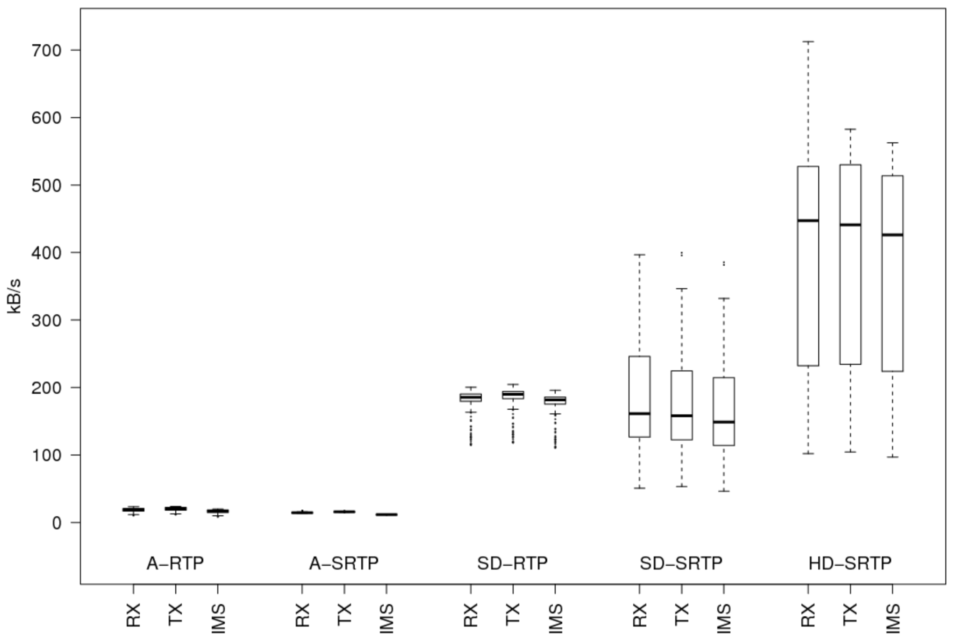

Figure 24: Throughput in kilobytes per second

based on the CPU load is predictable: plain audio calls generate less CPU load than video calls, and naturally HD video is more processor heavy than SD video calls. The difference between RTP and SRTP calls is also foreseeable: SRTP calls require more processing than the respective RTP calls. After all, in the RTP calls the packets are only relayed through, whereas in SRTP calls the packets are decrypted and encrypted. This behavior can be seen between the audio calls as well as between the SD video calls.

There is a little more variance in the CPU load in the SRTP video calls than in the other calls. This may be because of the greater variance in the data rate in SRTP video calls, as was seen in Figures 23 and 24.

Another interesting performance measurement is how much delay the Media Relay generates when handling the packets. The delays for the different call scenarios are presented in Figure 26. The delays are also divided into incoming and outgoing groups, which denote the direction that the packets are going. Incoming group here consists of the packets that arrive from the WebRTC client and which the Media Relay delivers towards the IMS. The incoming packets are possibly decoded from SRTP to RTP in case of an SRTP call. Outgoing group is the opposite, i.e. the packets which Media Relay transfers from the IMS to the WebRTC client. The outgoing packets are possibly encoded from RTP to SRTP in case of an SRTP call. Naturally, the SRTP conversion does not happen in the end-to-end RTP calls (A-RTP and SD-RTP).

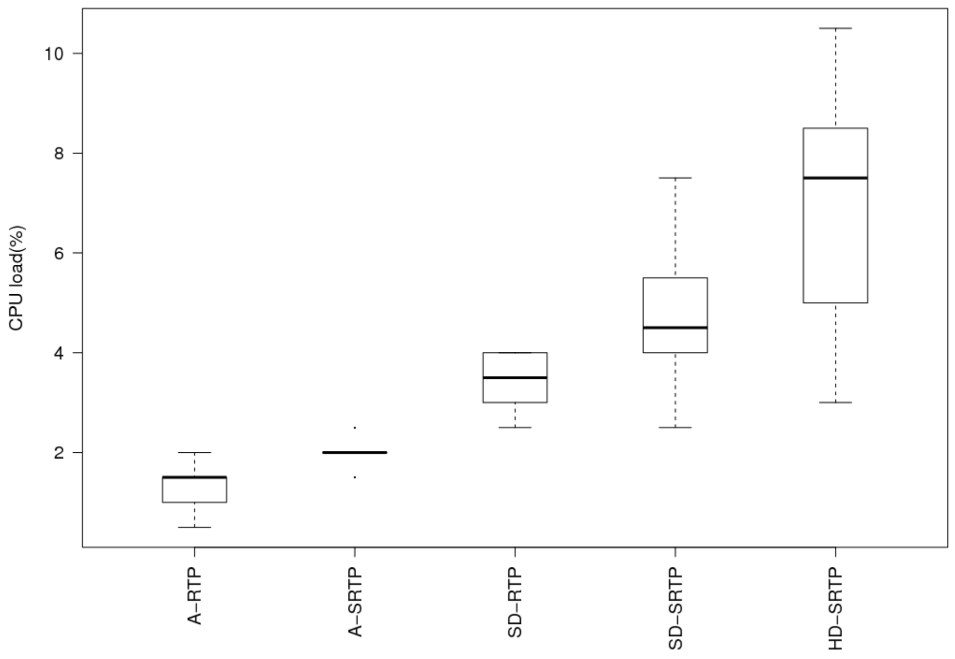

Figure 25: Single call CPU load

Vast majority of the delays are quite concentrated, which means that the box and the whiskers do not extend very far from each other. However, the outliers reach relatively far, in some cases above two milliseconds. Therefore the delays are presented on a logarithmic scale, so that the differences in the small delays remain visible.

As expected, the RTP calls have smaller delay than the respective SRTP calls, although the difference is not big. Both incoming and outgoing delays are smaller when the packets are only passed through, instead of doing the RTP-SRTP conversion which increases the delay. The greatest variance in the delay is clearly in the HD video call. This is interesting, since the payload sizes of the HD video packets are not much greater than the payload sizes of the SD video packets on SRTP calls, as shown in Figure 22. The HD call, however, has more variance in the throughput, as presented in Figures 23 and 24. High bursts of packets might cause higher delay for the last packets of the bursts.

One notable difference is also that the delays of outgoing packets are smaller than the delays of incoming packets, both in audio and video calls. The reason for this cannot be in the SRTP handling, since processing the outgoing packets is faster than processing the incoming packets also in the RTP calls, where Media Relay only passes the packets through. Since the differences are relatively small, there was no need to invest much time to figure out the root cause. Anyhow, more profound profiling would be needed to find out the actual reason behind this behavior.

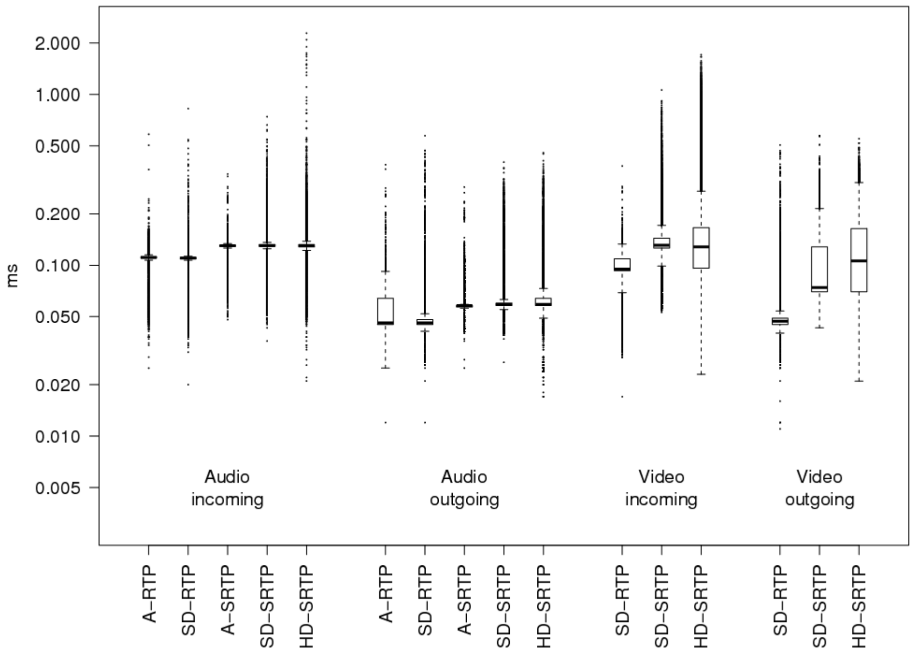

Figure 26: One-way packet delay

The measured delay is only the processing delay, e.g. how much it takes for Media Relay to receive a packet, operate on it, and send it forward. So this does not take into account the network delay that comes from routing and delivering the packet to and from the Media Relay.

As recommended in [26], the one-way end-to-end delay should be kept under 150 milliseconds if possible, and not exceed 400 milliseconds which is considered unacceptable. If Media Relay takes about 0.1 to 2 milliseconds, and it is done twice on one-way end-to-end call, the Media Relay is unlikely to become the bottleneck in single call scenarios.

4.3.6 Load Testing

Single call scenarios give a certain image of the performance characteristics between different types of calls. The Media Relay's processor load with a single call stays under 10 percent even in the case of an HD video call. To obtain an indication of the capabilities of the Media Relay we need to test the performance with several simultaneous calls.

For the load tests we set up ten client machines with two user accounts each. From every machine one HD video call was made to the other user on the same machine, through the Media Relay. Only one video call was made per machine, since generating, sending and receiving HD video turned out to consume significant amount of the client machines' CPU time—additional calls would have already slowed down the client processes, which could have affected the measurement results.

Each call was established after one another, with a minimum hold time of one minute between the calls. During this hold time the CPU performance of the Media Relay was measured during one minute, using a two second time window for the sampling.

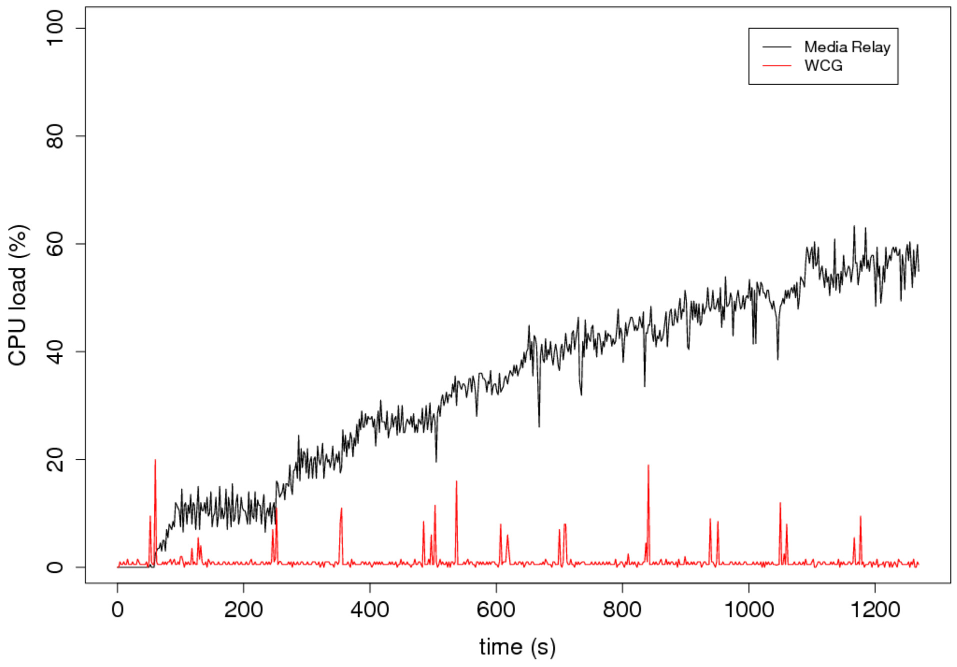

Figure 27: CPU load in load testing

Figure 27 shows the CPU loads of the Media Relay process and the WCG process during the whole measurement period. The Media Relay's CPU load is drawn in black, and the WCG's CPU load in red.

The moments of the call setups can be easily seen in the WCG's CPU load, indicated by the spikes in the otherwise quite idle process. It is the call setups when the WCG receives the call offer from the client and delivers it forward, and also when the Media Controller instructs Media Relay to reserve resources for the call. During the calls the WCG does not need to do much anything.

The different calls can also be observed in the Media Relay CPU load, as it increases after the spikes in the WCG's CPU load. Ten consecutive HD video calls raise the Media Relay's CPU load to around 60 percent.

The average CPU loads of the cumulative calls are presented in Figure 28. The CPU loads were measured during the one minute hold times between the calls.

The first call generates a bit higher load than in the single call measurement presented in Figure 25. There is also some variance between the differences of the

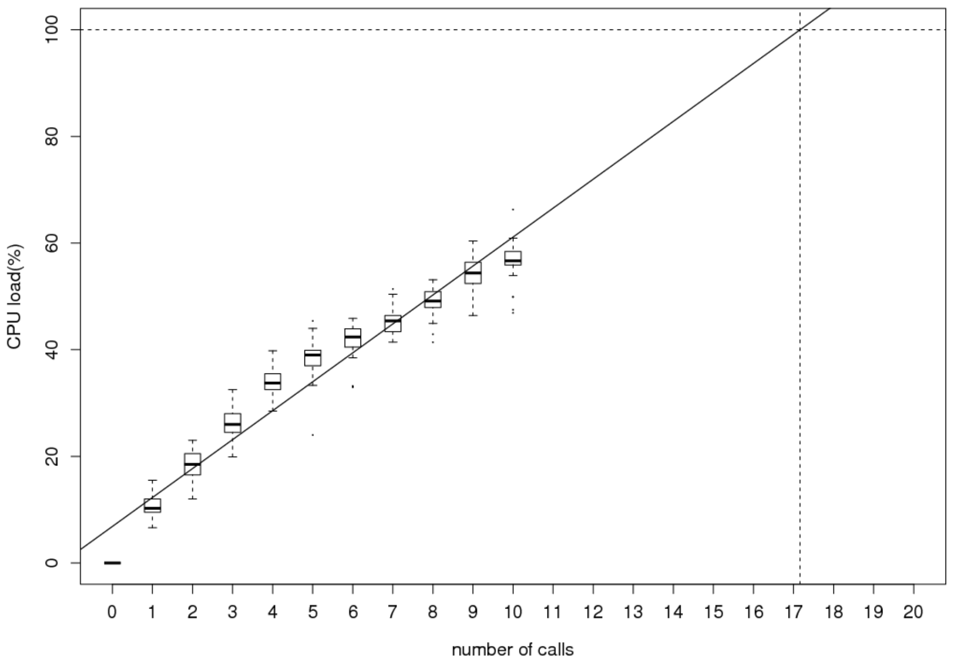

Figure 28: CPU load per number of calls

average CPU loads between the consecutive calls. These are probably caused by overall fact, that the variance in the bitrate of HD video calls is relatively high. It probably also depends on the client machine and the available resources for creating and sending the HD video. So even though the client machines were all modern PCs and all calls were made with an up to date Chrome as the client browser, the individual calls differ in the needed processing requirements.

Figure 28 also shows the linear trend of the CPU loads for the cumulative HD video calls. The trend line reaches 100 percent a little after 17 calls. Even though some of the individual CPU load measurements differ quite a bit from the trend line, we can say that based on this measurement, one Media Relay can theoretically handle maximum of 17 simultaneous HD video calls. The performance, and therefore the media quality, might of course degrade when approaching 100 percent CPU load. This said, a little safer estimate for the maximum number of simultaneous HD video calls would be around 15.

Figure 29 presents the Media Relay CPU load measurement samples as a function of all the sent (TX) and received (RX) data on all interfaces.

Like in the previous graph, the trend is visibly linear. The data itself seems to fit even better to the linear trend than in Figure 28, because here the different calls are not separated. This graph gives a better overall estimation of the Media Relay capacity, since it does not depend on the call type. The linear trend reaches the maximum CPU performance at 30 791.82 kilobytes per second, which is the

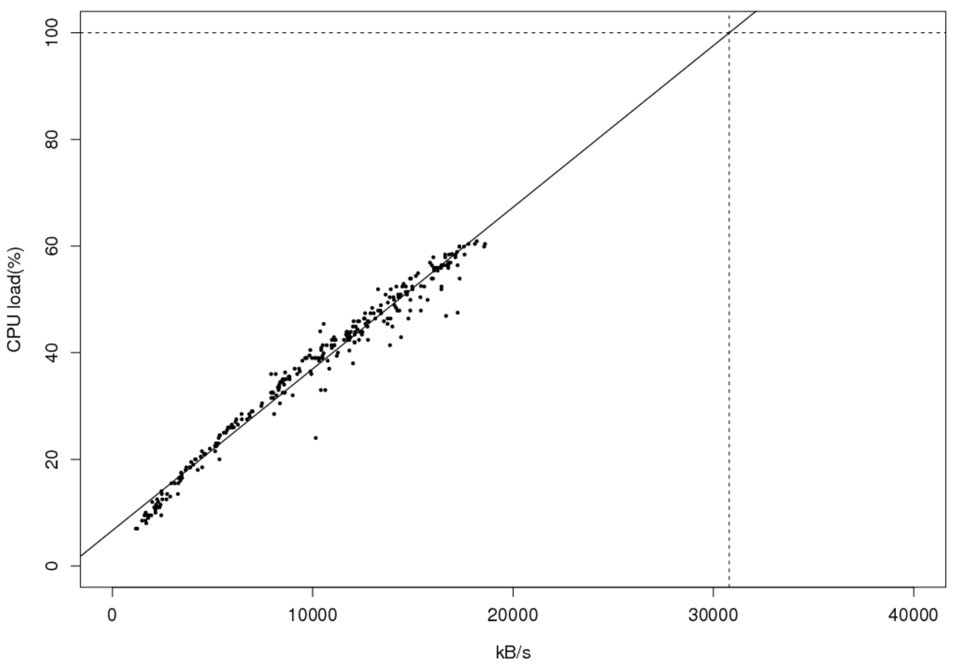

Figure 29: CPU load as a function of throughput in HD video load testing

theoretical maximum throughput for HD video calls. This presentation seems to be fairly good for estimating the overall performance.

Since the video packet payload sizes in the SD video calls are very similar to the payload sizes of the HD video calls, it is fair to assume that this model applies well to SD video calls also. In audio calls, however, the payload sizes in the packets are significantly smaller. This probably causes more processing overhead when comparing to the video calls. Therefore we conducted another load test, using only audio calls. The test was identical to the video call load test: a total of ten audio calls were setup one after another, with one minute hold time after each new call.

Figure 30 presents the CPU load measurements as a function of total traffic in Media Relay for ten cumulative audio calls. Even though the data set is quite small, and the overall performance reaches barely ten percent, we can see from the trend line that the maximum throughput for audio calls is significantly smaller than for video calls: hundred percent CPU load is achieved already at 4 063.57 kilobytes per second.

Another important topic also in load testing is the delay: does the delay increase to unacceptable lengths when Media Relay has to handle greater number of packets? In Figure 31 we see the delays of the audio packets for the cumulative HD video calls. Once again, the delays concentrate around small values, while the individual maximums reach quite high. Therefore the delays are presented on a logarithmic scale.

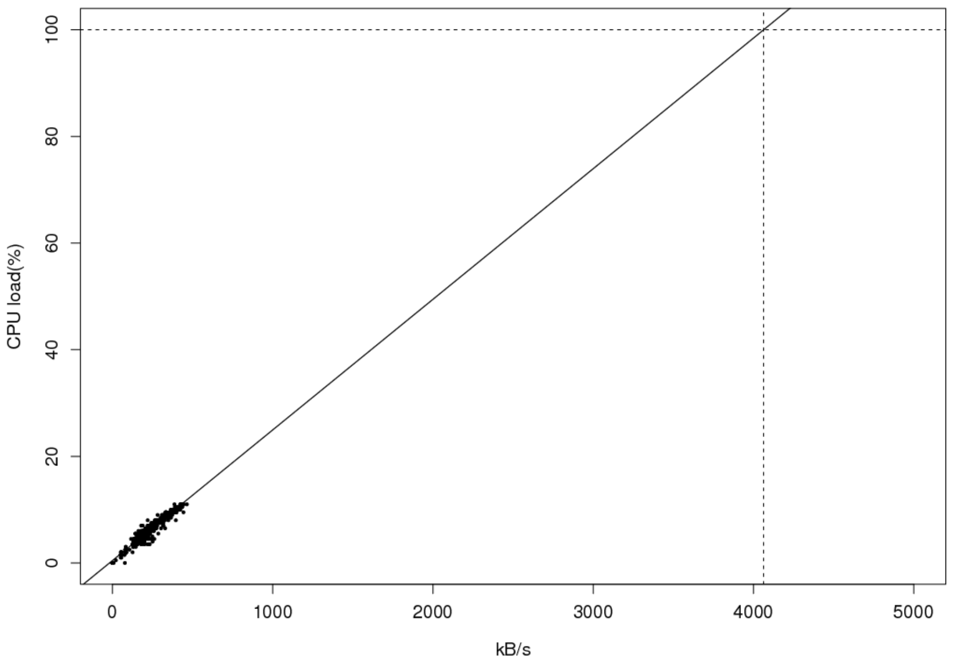

Figure 30: CPU load as a function of throughput in audio load testing

Evident from the graph, the median delay does not increase much when adding calls on top of one another, which is good from the user experience perspective. Even with ten simultaneous HD video calls the median delay of audio packets remain a little over 0.1 milliseconds.

However, the outliers show that the maximum delay increases quite rapidly when there are more traffic to be handled. The maximum time an audio packet spent in Media Relay was 8.17 milliseconds, when there were eight simultaneous calls ongoing. If we consider the recommended maximum end-to-end delay of 150 [26], occasional delays of maximum eight milliseconds should not be harmful. Even if the audio packet goes through the Media Relay twice in case of a call between two WebRTC clients, it should hit the maximum in both of them, and the delay would still probably be under 20 milliseconds. Of course the total end-to-end delay depends greatly on other devices in the network.

When investigating the delays of the video packets during load testing, we see even greater variance than with the audio packets. Figure 32 shows the delays of the HD video packets for the cumulative calls, also presented on a logarithmic scale.

What is positive, is that the median delay stays between 0.1 and 0.2 milliseconds with all ten calls. But in with the video packets, the maximum delays range even further than with the audio packets. The single maximum delay is reached with nine simultaneous calls, when a packet spent 38.00 milliseconds in the Media Relay.

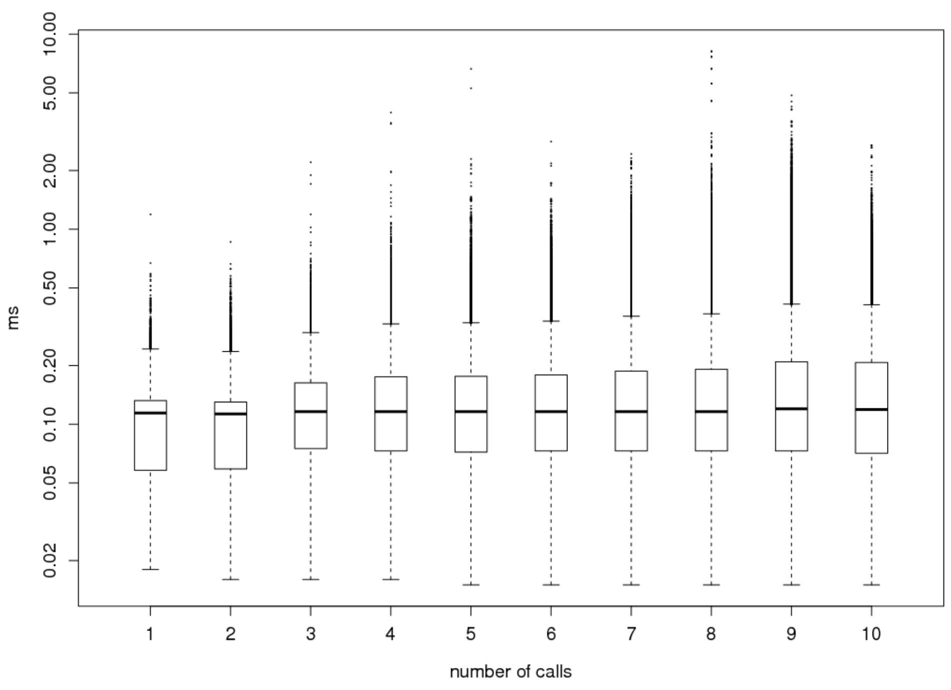

Figure 31: One-way packet delay of audio packets in HD video load testing

Delays reaching close to 40 milliseconds per one Media Relay traverse could already be notable in the end-to-end delay, if they would occur more often. As individual peak points they should not, however, degrade the overall quality too much. Also, taking the 95th percentile of the ninth call reveals that majority of the delays still lie below 0.98 milliseconds.

Media Relay does not perform any synchronization between the audio and the video streams. The synchronization is then up to the clients. This means that when the individual delays might increase as the number of calls grow, and because video delays increase more rapidly, the audio packets pass through Media Relay faster than the video packets. When the client synchronizes the audio and the video streams, it is likely the video stream then that dictates the overall delay, no matter how fast the audio packets had arrived.

4.4 Measurement Analysis

This section analyses the measurement results presented in the previous section. The emphasis is on the performance of the Media Relay, focusing on the CPU load and the delay that Media Relay introduces to the media traversal. First we analyze the characteristics of the different traffic profiles. The CPU load analysis leads us to estimate the maximum number of calls Media Relay can handle. After analyzing the delays, we discuss the generality of the measurements.

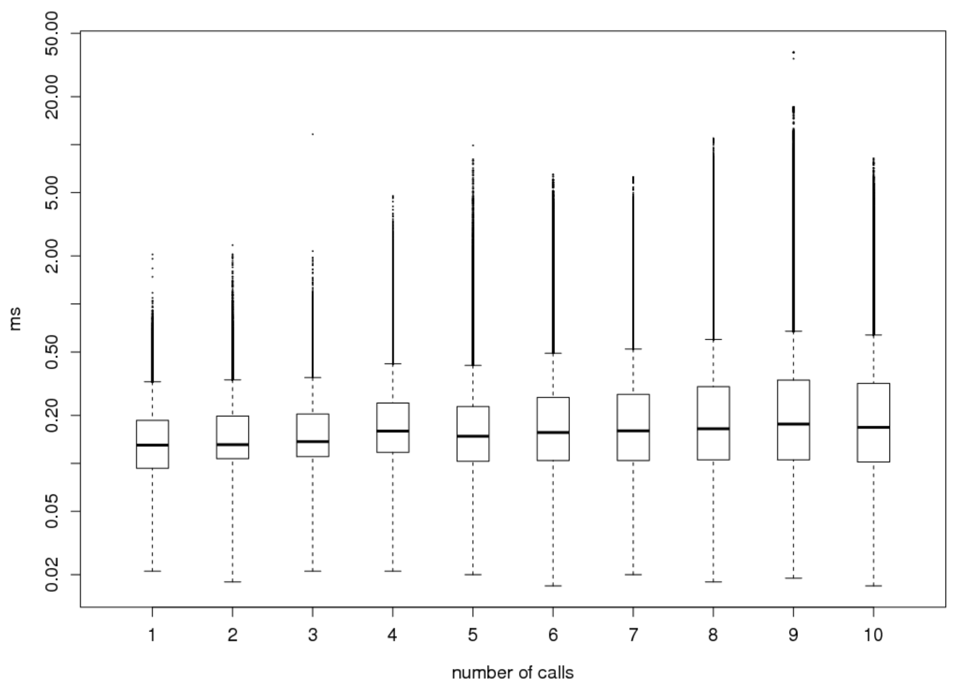

Figure 32: One-way packet delay of video packets in HD video load testing

4.4.1 Traffic

The measurements included different traffic profiles, ranging from plain audio calls to HD video calls, and containing both unencrypted (RTP) and encrypted (SRTP) calls. The RTP calls as such are not that relevant as a real-life use cases, since the WebRTC specifications dictate that all media has to be encrypted in a WebRTC session. Also, the current browsers already implement SRTP capabilities, and do not even offer unencrypted WebRTC possibilities. Nevertheless, the RTP measurements were conducted alongside the SRTP measurements, since the original setup consisted of Media Relay without SRTP encryption and a customised browser supporting unencrypted WebRTC. Therefore the measurements were also included in this thesis for reference.

Because of the different browsers used for the measurements, the results have somewhat different characteristics between the RTP and the SRTP calls. The nature of the different codecs pose some constraints to the analysis of the different call scenarios, but for general measurements that aim to provide some indication of the Media Relay's performance, the differences are not that significant. Nonetheless, they provide another perspective on how the system behaves with different setups.

As mentioned in Section 2.1 there is an ongoing debate about the mandatory to implement video codec between VP8 and H.264. Comparing the payload size results in Figure 22 between the SD-RTP (H.264) and SD-SRTP (VP8) calls we can see that there is some, but not much difference. The average payload size of the video packets is really close between the two. VP8 has a little more variance in its payload sizes, whereas H.264's minimum payload sizes reach a bit closer to zero.

The measured packet rates in Figure 23 are also very close to each other in SD-RTP and SD-SRTP, when looking at the median values. In SD-SRTP there is a bit more variance, whereas in SD-RTP the packet rate stays really close to the median. All this is of course dependent on the codec implementation. The Leif browser we used has Ericsson's own implementation of H.264, and the VP8 in Chrome is the browser's own native implementation.

The performance differences between VP8 and H.264 cannot be directly interpreted from the measurements presented in the previous section, since all the calls with VP8 included SRTP encryption and decryption, whereas H.264 calls were plain RTP, going straight through Media Relay. In Media Relay's point of view that would also be the only difference between the two, since Media Relay does not care about the payload type inside the (S)RTP packets.

These are only small observations about the different codecs noted in our measurements, and strong statements cannot be made based on them in favor of one or the other. There are more important factors to be considered and precise studies to be made when deciding which codec will eventually be more feasible for WebRTC.

4.4.2 CPU Load

As a prototype system, Media Relay is not expected to handle hundreds or thousands of simultaneous calls. Even a handful of calls is enough to trial the functionality and to use the prototype in a proof of concept scenarios.

The different call scenarios have different performance characteristics. As Figure 25 displays, the more data and complexity (as in case of encryption) is present in the call, the more processing it requires. This comes as no surprise. Even though the SD-RTP video call is on par with the SD-SRTP video call when it comes to the RTP packet payload sizes, the SRTP conversion is a big factor in the processing requirements. HD video calls naturally add even more data to be processed.

Audio calls and SD-RTP video calls each fall into a quite narrow CPU load range, whereas the SRTP video calls, both SD and HD, have a little more variance in the CPU load they cause in Media Relay. The variance is probably caused by the higher variance in the packet rate, when comparing to the SD-RTP calls.

The CPU load for all types of single calls remains mostly under ten percent, even for the HD video calls. Single audio calls generate only one to two percent CPU load. These kind of measures seem good enough for a prototype system, since in terms of CPU load the Media Relay appears to be able to handle even tens of simultaneous calls.

This was further validated in the load tests, where Media Relay had to handle ten simultaneous HD video calls. The overall CPU load of the Media Relay process increased to almost 60 percent. The results were even better than expected, and the estimated maximum of 15 simultaneous HD video calls for a prototype system sounds promising.

Presenting the load measurements as a function of total throughput in Figure 29 gives a more generic picture of the Media Relay's capabilities. In Section 4.3.4 it was estimated that a single HD video call generates traffic on average around 1 700 kilobytes per second. This seems to be in line with the load measurements, where ten simultaneous calls reach maximum 18 599.71 kilobytes per second.

The CPU load as a function of throughput seems like a good estimate on the overall capacity of Media Relay, since the data fits well to the linear regression. This kind of model, however, depends on the traffic profile, since different types of data have different processing overhead.

Estimating the actual maximum number of calls Media Relay can handle is quite hard, since the throughput, and also the CPU load of single calls can vary quite a bit, as we have seen in the measurement results. We can try to give rough estimates based on different throughput levels, but it is worth noting, that for example bursts in traffic increase the processing load momentarily. This could limit the actual number of possible simultaneous calls, or at least affect the media quality.

Table 9 in Appendix A presents the measurement data of the call bitrates in kilobytes per second. The data is visualised in Figure 24. Using this data we can calculate approximate estimates on the maximum number of calls that Media Relay can handle. The calculations are based on the estimated maximum bitrates that Media Relay can handle, presented in Section 4.3.6: for video calls it is 30 791.82, and for audio calls it is 4 063.57 kilobytes per second. The maximum bitrate is then divided by the total traffic $RX + TX + (2 \times IMS)$ for each call type, presented in Section 4.3.4. IMS is counted twice, because the RX and TX on the IMS side was always equal, and thus was presented in the table and in the graph only once.

In Table 5 we present the maximum number of calls for the actual real-life use cases: encrypted audio call, and encrypted SD and HD video calls. The maximum number of calls are calculated using three different values for the total traffic: median, third quartile and maximum. All the results are rounded down to the closest integer.

Table 5: Estimated maximum number of calls for different bitrate limits

Call	**Median**	**3$^{\text{rd}}$ Qu.**	**Max.**
A-SRTP	75	74	69
SD-SRTP	49	34	19
HD-SRTP	17	14	12

The values based on the maximum bitrates are safe estimates on how many calls Media Relay could still handle, if all the calls run on maximum bitrate all the time. For HD video calls this yields 12, which certainly should not overload Media Relay—with ten HD video calls we reached only 60 percent CPU load in our tests.

The Median values are a bit more optimistic. If the CPU is running on hundred percent load when the calls run on median bitrates, any bursts or variance might easily degrade the media quality. For HD video calls this estimate is 17, which is in line with the estimate we got already in Section 4.3.6.

The third quartile estimates are probably the best estimates to be applied for real-life scenarios: they leave a little margin for the extreme cases, yet providing quite good estimates for the maximum capabilities of Media Relay. By this estimation, a single Media Relay could handle 14 simultaneous HD video calls.

What comes to SD video calls, the estimates spread a bit more: the maximum number of calls is somewhere between 19 and 49, the third quartile estimation giving 34 calls. The estimates for maximum number of plain audio calls fall quite closely around 70 calls.

4.4.3 Delay

Delay plays an important role in real-time applications. Like [26] states, with one-way delays under 150 milliseconds, most real-time applications will experience essentially transparent interactivity. Delays between 150 and 400 milliseconds can be tolerated, without making an excessive number of user experiences unacceptable.

The measurements concentrate only on the delay that Media Relay introduces to the media transmission, which can be valuable information when planning a network for WebRTC communication.

The delays in single call scenarios at Media Relay are minor. The maximum delays reach to a few milliseconds, while the average delays remain well under one millisecond. These kind of figures do not pose any threats to the recommended 150 millisecond limit.

The difference between the RTP and the SRTP calls is not major, but clearly noticeable. While the packets in pure RTP calls are only passed through, the packets in SRTP calls need to be encrypted or decrypted, which naturally increases the processing load and thus also the delay.

However, there seems to be only little difference between the audio packet delays and the video packet delays, even though the average sizes of the video packets are roughly ten times larger. The overall load in the single call scenarios stays so low, that the difference in the payload size does not result in noticeable differences in the processing time.

When it comes to increasing the load in Media Relay, the differences between audio packet delays and video packet delays become more apparent. With ten calls, the audio packets still travel through Media Relay quite quickly, with maximum delay in the measurements being under ten milliseconds. For the video packets, the maximum delay reached nearly 40 milliseconds. This shows that when the load in Media Relay grows, the larger video packets start taking a bit longer to be processed. However, it is only the maximum delays that span rapidly; for example the third quartile, and even the higher whiskers in the boxplots grow less than one millisecond.

Even when Media Relay load increased close to 60 percent with ten simultaneous HD video calls, the median delays of both audio and video packets remained below 0.2 milliseconds. This is good in terms of media quality, and gives confidence when planning networks so that the one-way end-to-end delay would most of the time stay under 150 (or at least under 400) milliseconds, as recommended.

4.4.4 Generality of the Measurement Results

The measurements were performed inside the company network, which has an average RTT less than one millisecond. This is probably much less than in the public Internet. Even though we did not measure any end-to-end delays or such, a more congested network could possibly affect the bitrate of the calls. Higher delays probably cause more variance in the bitrates. Also higher packet loss could cause more data to be sent, because the video codecs may need to request missing key frames, which are generally larger than any adjacent frames.

Also, due to lack of proper load testing environment, the generalizations of the estimates presented in the whole Section 4.4 are pretty broad. The number of executed test calls was relatively low, and especially the load tests were not trivial to repeat due to manual testing. However, other single call test executions gave similar results as documented in the thesis results. In addition to the generalization, it is difficult to estimate how high the load would need to rise, so that it would already start to affect the quality of the ongoing calls.

The audio and video quality were not considered in this thesis, other than analysing the delay. Mostly both the audio and the video ran smoothly, but at times there were some glitches and freezing frames, a bit more so in the load testing than in the single calls.

The client software in all the measurements was Chrome (or Leif, modified Chrome, in the RTP calls). At the moment also Firefox supports WebRTC, but in the future, the client could well be any other software, that has different characteristics and different codec implementations. Also the actual video resolution can be specified more precisely in the JavaScript application—it is not restricted to only two options.

4.5 Summary

We used a prototyping environment to examine the performance characteristics of the Media Relay. The prototyping environment consisted of a simple network with one Media Relay, which operated alongside the WCG and the SDS as the simulated IMS network. As the client software we used Chrome and Leif browsers.

A series of measurements were run with five different traffic profiles, containing unencrypted and encrypted media, both audio and two types of video. After analyzing the traffic profiles, we focused on the Media Relay's performance metrics, in terms of CPU load and delay.

The measurement results revealed that encrypted audio calls run on roughly two percent CPU load, while encrypted SD video calls generate a bit over four percent CPU load per call. The unencrypted versions of those calls generate on average 0.5-1 percent less CPU load. Encrypted HD video calls generate close to 8 percent CPU load, with relatively higher variance than the other calls.

The average delays in all the call types were less than 0.2 milliseconds, when measuring single calls. What was interesting, was that the delay of incoming packets (from the WebRTC client towards the IMS) was explicitly higher than the delay of outgoing packets (from the IMS towards the WebRTC client). This was even with unencrypted calls, so it was not a result of possible difference in encoding and decoding processes.

By running ten simultaneous calls, cumulated on top of each other, we obtained some load performance statistics for encrypted HD video calls and encrypted audio calls. From the video calls we discovered, that the median delay did not grow almost at all, even with ten simultaneous calls. The maximum delay did grow for both audio and video packets, but that concerned only a small fraction of packets.

From the load statistics we derived estimates on how many simultaneous calls Media Relay could handle. The estimates are based on the performance measurements and the measured bitrates for the different calls. We presented three different estimates: an optimistic estimate based on the median bitrates, an estimate with a small margin based on the third quartile bitrates, and a safe estimate based on the maximum bitrates.

5 Discussion

This chapter discusses how Media Relay could perform in real-life scenarios, still keeping in mind that it is merely a prototype system. We also present some ideas for future work on how Media Relay could be improved, and what sort of measurements could still be conducted to more thoroughly characterize the performance of it.

5.1 Media Relay as a WebRTC Gateway

Media Relay acts as a gateway between the public Internet and the private network, owned by for example an operator. It converts a WebRTC call to a traditional IP-based multimedia call, by terminating the DTLS and ICE sessions and decrypting the forwarded RTP media, and vice versa. Every time a new intermediate node like this is introduced to the media path, some amount of extra processing and delay is bound to happen.

In real-time applications delay is a crucial element: a delay too high will render the service practically unusable. Some limits for a "too high" delay are proposed in [26]. The delay depends greatly on the processing that needs to be done in the intermediate node. In this case, there existed a basic relay node, to which we implemented the security functionality. This consisted of DTLS-SRTP protection, coupled with ICE, as was required in the WebRTC specifications.

As SRTP encryption and decryption was considered computationally heavy operations, we measured the effect of the new functionality compared to the initial state. Also the overall performance was measured to see how Media Relay would perform in different scenarios.

In Section 4.4.2 we presented estimates for the maximum number of simultaneous calls that a single instance of Media Relay could be able to handle. Around 14 simultaneous HD video calls is not much, certainly not product level capacity. But for a prototype system, which Media Relay is, this is certainly acceptable. Also, since this is a software solution, the capacity scales according to the hardware in use. If the hardware does not support OpenSSL, the capacity is probably lower.

The measurements were performed in a restricted and fairly stable environment. In the public Internet conditions probably vary significantly more, which naturally could affect the Media Relay's performance. Also the traffic scenarios we tested were very simple. In real life this kind of system would handle varying mixtures of different kind of traffic. The audio calls compose of relatively steady stream of small packets, whereas the video packets are distinctly larger and the bitrates of the video streams fluctuate much more.

The overall capacity is a function of the traffic mixture: the different call types require different amount of processing per kilobytes per second, because the different sizes of packets introduce varying overhead in the processing.

All the measurements were performed with a single Media Relay in the network setup. In a WebRTC to WebRTC call, this causes the media to pass through the same Media Relay twice: once when it enters the network, and once when exits the network. In a more complex scenario there could be more instances of the Media

Relay running on different access points of the network. In these cases the media would travel through two different Media Relays, each once. This naturally divides the processing requirements per Media Relay roughly by two. The delay per Media Relay would still be the same as was measured in Section 4.3.5, since we measured one-way delay per each direction.

5.2 Future Work

As a prototype system, Media Relay is not rock solid. Therefore, there are a lot of things that can be improved. In addition to actual functional improvements, Media Relay could benefit from a proper testing environment, which would also come useful in measuring the performance more thoroughly.

Since the objective of Media Relay's implementation was to get "something functional", performance was not on the top of the wish list. As mentioned, Media Relay is single-threaded. So in order to fully utilize today's modern multi-core hardware, multiple instances of Media Relay need to be executed simultaneously. This is both cumbersome and hard to scale. One obvious improvement would be to make Media Relay multi-threaded. This could, however, require quite a lot of redesigning, since if multi-threading is not considered already in the very beginning, it can be difficult to integrate into a software due to possible architectural changes.

Furthermore, there are certainly places where the code can be optimized to gain better performance. The adopted libraries look fairly well written on the surface, but nevertheless, proper profiling could easily point out bottlenecks both in the library code and the application code.

A proper testing framework would benefit both implementation and measuring the Media Relays performance. Now, due to lack of proper feature test framework, the test runs in the measurements were done manually. This means firing up two browser windows or tabs for each call, and doing all the call setup steps by hand. In a proper test framework the tests could be run automatically during the implementation, and the measurements could be performed more extensively and systematically.

Creating the signaling parts for test calls would probably not be a big effort, but generating the video and audio streams, and sending them in SRTP could require a bit more work. Certainly there are programs and libraries for all those tasks, so it is only a matter of time to be used, that this kind of system could be built. Also, for the load testing, the test framework should be able to deal with N amount of, say, HD video calls, without hogging all the resources.

In this thesis the load tests consisted of ten simultaneous calls. In case of HD video calls this was enough to increase the Media Relay's CPU load a little over 60 percent. But for audio calls the CPU load reached hardly ten percent. Bigger data sets would yield more accurate results, especially in load testing. Also the variance in the results could be defined more specifically with higher volume in the test data.

Since our measurements only included simple traffic scenarios, more testing would be needed to thoroughly understand how different traffic mixtures affect the Media Relay's performance. These traffic mixtures could, for example, be obtained from actual client networks, and then simulated in the testing framework. In addition to that, various network scenarios could be measured by simulating congestion and packet loss.

Another interesting characteristic in performance measurements is call setup rate. It is feasible to know, how many calls can be set up per second. This is greatly affected by the handling of the initial connectivity setups, i.e. the STUN messages for the ICE, and the DTLS handshake before the actual data can start to flow.

All the basic WebRTC functionality were implemented to Media Relay, but some interesting features could still improve the interoperability and user experience.

BUNDLE feature was introduced in Section 3.4.3. The feature aims to bundle the audio and video streams into one port, thus saving resources and making for example NAT and firewall traversal easier. BUNDLE could be easy to implement, since it has some similarity with rtcp-mux function, which was now implemented. The BUNDLE feature is used in Chrome by default, so it would definitely come to use in browser-based WebRTC communication.

One significant interoperability feature is transcoding. Especially when connecting browsers to IMS clients, there may very well be a mismatch in the offered codecs. With the MTI video codec still being undecided, even the specification do not yet tackle the issue. VP8 and Opus, which both are supported in the major browsers Chrome and Firefox, might not be that widespread on the IMS side clients and services. At the time of writing this, Media Relay has already gained video transcoding support between VP8 and H.264. This was implemented as a separate service, which can be added to Media Relay when needed. It can be hosted on a separate device, so that the heavy transcoding process can be performed on a dedicated hardware.

The transcoding functionality could be further extended by adding support for more codecs in addition to VP8 and H.264. For example transcoding between Opus and PCM could be useful in some cases, even though PCM is already mandatory part of WebRTC and very widely used in all voice communication. Also, the next generation video codecs are making their way to provide better quality and higher compression: VP9 [27] as the successor of VP8, and H.265 [28] as the successor of H.264.

As we learned in Section 2.6.1, TURN offers another additional way to traverse through restrictive networks, by relaying data through trusted servers. Incorporating a TURN server by the Media Relay would make Media Relay more usable for example on the edges of strict corporate networks. This is already being investigated, and the implementation is left for future work.

5.3 Summary

Media Relay was implemented as a prototype system, and thus the performance was not the main objective. In order to proof that the concept works, it was enough that some simultaneous calls would go through without problems. Nevertheless, the measurements proved that Media Relay performed fairly well. The delay remained moderate, and in terms of performance Media Relay met the expectations.

Based on the measurements alone, however, it is difficult to say how Media Relay would perform in real-life scenarios. The network conditions in the Internet are not as good as in the test network. The traffic mixture in real-life is also not as homogeneous as in our load tests.

To improve Media Relay, several future work items were introduced. Media Relay's architecture could be improved and additional features could be implemented. Creating a proper testing framework and test automation would help to further measure and analyze the Media Relay's performance.

6 Conclusions

A joined standardization effort by IETF and W3C has been emerging for the past few years, trying to specify means for real-time peer-to-peer multimedia communication between web browsers. The effort has been adopted by some of the major browser vendors, who have been integrating the functionalities to their software in a rapid pace. Basic audio and video call applications are already available, with the addition of data transfer features.

In the dawn of native web-based multimedia communication, Ericsson started to design a gateway system, that could connect the new WebRTC functionalities in the Internet with the existing IMS networks. As a result, Ericsson implemented MMTelGW: a prototype system, that was planned to do the necessary conversions in the signaling and the media planes, so that the two worlds would interact seamlessly.

Since even the specifications were not ready, the prototype system was first implemented without security: the media was unencrypted, and for testing purposes even a specific browser was devised based on Google's Chromium browser.

In order to comply with the standards, and to be able to interact with current WebRTC compliant browsers, we implemented the necessary security features to the Media Relay, an integral part of the MMTelGW. This included the data encryption in form of DTLS-SRTP, and the connectivity establishment by using ICE. The implementation was successfully tested with Chrome and Firefox, being the two major browsers which support WebRTC.

To evaluate the usability of Media Relay as a prototype system we conducted a series of measurements, using Chrome as the client software. The measurements included encrypted audio calls, and encrypted video calls with low and high resolution video. We also included measurements with unencrypted audio calls and unencrypted low resolution video calls, to evaluate the security functionalities' effect to Media Relay's performance.

The processing load of encrypted calls is naturally higher than the processing load of corresponding unencrypted calls. The latter is only matter of forwarding the RTP packets, whereas the former adds the encryption and decryption procedures to the relaying process.

Based on the measurement results we presented estimates on the Media Relay's processing capabilities for the real-life use cases, i.e. the encrypted WebRTC calls. Since the effect of hundred percent CPU load to the ongoing calls was not evaluated, we present the estimates that have reasonable margins not to overload the Media Relay process. That said, we can state that a single Media Relay instance can handle maximum of 14 simultaneous HD video calls between two WebRTC peers, when both clients are connected to the same Media Relay. The same figure for SD video calls was estimated based on the HD video call data, resulting in 34 simultaneous calls. Audio call measurements suggest that 74 simultaneous audio calls can be made without pushing the Media Relay to the limits.

In addition to the Media Relay's CPU load, we measured the delay that Media Relay introduces to the media path. The results state that for single calls the (S)RTP packet processing delay remains mostly under one millisecond. For real-time communication applications this kind of delay is certainly acceptable. When the number of simultaneous calls increases, the maximum delays seem to increase too: for ten simultaneous calls the maximum delay for video packets reached 38.00 milliseconds, and for audio packets the maximum delay was 8.17 milliseconds. Even though the high end of the delay distribution raised quite a lot, it was only a fraction of the packets that spent over five milliseconds in the Media Relay. Even under heavy CPU load, the media delays for both audio and video packets remained very steadily around 0.1 milliseconds.

To serve its purpose as a prototype system, Media Relay fills its expectations. A set of even tens of simultaneous calls is enough to test WebRTC functionalities in different scenarios. Being compatible with both Chrome and Firefox grants a big share of the browser base. Now that the standards are still evolving, interoperability is not automatically guaranteed. But over time, when both the standards and the implementations mature, it should be indisputable that any standard compliant application should be able to successfully interact with one another.

In addition to the measurement analysis, we present a set of future work items that can be addressed to improve the Media Relay. Some features, such as TURN or BUNDLE, would guarantee better performance or more seamless interoperability with different browsers and client software. Other improvements, such as multi-threading, aim to enhance the implementation and evaluation of the Media Relay itself.

The emerging WebRTC standardization itself encourages innovations in the web communication area. A WebRTC gateway system, such as the combination of Media Relay and WCG, opens up new possibilities to further extend the new technology. Combining the novel WebRTC capabilities with legacy telephony systems and other services opens up new opportunities for the stakeholders: the users, the developers, as well as the service providers have the possibility to compose a whole new set of applications that one can only imagine.

References

[1] Alvestrand, H., *Overview: Real Time Protocols for Browser-based Applications*, draft-ietf-rtcweb-overview-06 (work in progress), February 2013

[2] Burnett, D. C., Bergkvist, A., Jennings, C., Anant Narayanan, A., *Media Capture and Streams*, W3C Working Draft, May 2013

[3] Bergkvist, A., Burnett, D. C., Jennings, C., Narayanan, A., *WebRTC 1.0: Real-time Communication Between Browsers*, W3C Working Draft, August 2012

[4] Valin, JM., Bran, C., *WebRTC Audio Codec and Processing Requirements*, draft-ietf-rtcweb-audio-01, November 2012

[5] Alvestrand, H., Grange, A., *VP8 as RTCWEB Mandatory to Implement*, draft-alvestrand-rtcweb-vp8-01, February 2013

[6] Burman, B. et al., *H.264 as Mandatory to Implement Video Codec for WebRTC*, draft-burman-rtcweb-h264-proposal-01, February 2013

[7] OpenH264, `http://www.openh264.org/` Referenced on May 14, 2014

[8] Uberti, J., Jennings, C., *Javascript Session Establishment Protocol*, draft-ietf-rtcweb-jsep-05, October 2013

[9] Handley, M., Jacobson, V., Perkins, C., *SDP: Session Description Protocol*, RFC 4566, July 2006

[10] Rosenberg, J., Schulzrinne, H., *An Offer/Answer Model with the Session Description Protocol (SDP)*, RFC 3264, June 2002

[11] Schulzrinne, H., Casner, S., Frederick, R., Jacobson, V., *RTP: A Transport Protocol for Real-Time Applications*, RFC 3550, July 2003

[12] Rescorla, E., *WebRTC Security Architecture*, draft-ietf-rtcweb-security-arch-07, July 2013

[13] Perkins, C., Westerlund, M., Ott, J., *Web Real-Time Communication (WebRTC): Media Transport and Use of RTP*, draft-ietf-rtcweb-rtp-usage-06 (work in progress), February 2013

[14] Baugher, M., McGrew, D., Naslund, M., Carrara, E., Norrman, K., *The Secure Real-time Transport Protocol (SRTP)*, RFC 3711, March 2004

[15] Dierks, T., Rescorla, E., *The Transport Layer Security (TLS) Protocol Version 1.1*, RFC 4346, April 2006

[16] Rescorla, E., Modadugu, N., *Datagram Transport Layer Security*, RFC 4347, April 2006

[17] Fischl, J., Tschofenig, H., Rescorla, E., *Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)*, RFC 5763, May 2010

[18] McGrew, D., Rescorla, E., *Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)*, RFC 5764, May 2010

[19] Rosenberg, J., *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols*, RFC 5245, April 2010

[20] Rosenberg, J., Mahy, R., Matthews, P., Wing, D., *Session Traversal Utilities for NAT (STUN)*, RFC 5389, October 2008

[21] Schulzrinne, H., Casner, S., *RTP Profile for Audio and Video Conferences with Minimal Control*, RFC 3551, July 2003

[22] Perkins, C., Westerlund, M., *Multiplexing RTP Data and Control Packets on a Single Port*, RFC 5761, April 2010

[23] Westerlund, M., Perkins, C., Lennox, J., *Sending Multiple Types of Media in a Single RTP Session*, draft-ietf-avtcore-multi-media-rtp-session-04, January 2014

[24] Holmberg, C., Alvestrand, H., Jennings, C., *Multiplexing Negotiation Using Session Description Protocol (SDP) Port Numbers*, draft-ietf-mmusic-sdp-bundle-negotiation-05, October 2013

[25] Camarillo, G., Schulzrinne, H., *The Session Description Protocol (SDP) Grouping Framework*, RFC 5888, June 2010

[26] International Telecommunication Union, *One-way transmission time*, ITU-T G.114, May 2003

[27] Grange, A., Alvestrand, H., *A VP9 Bitstream Overview*, draft-grange-vp9-bitstream-00, February 2013

[28] International Telecommunication Union, *High efficiency video coding*, ITU-T H.265, April 2013

A Measurement Result Data for Boxplots

The following tables present the boxplot data values (median, first and third quartiles, and whiskers), as well as minimum and maximum values.

Table 6: Payload size of audio packets in bytes, Figure 21

Call	**Dir.**	**Min.**	**Lower whisker**	**1st Qu.**	**Median**	**3rd Qu.**	**Higher whisker**	**Max.**
A-RTP	IN	160	160	160	160	160	160	160
	IMS	160	160	160	160	160	160	160
	OUT	160	160	160	160	160	160	160
SD-RTP	IN	160	160	160	160	160	160	160
	IMS	160	160	160	160	160	160	160
	OUT	160	160	160	160	160	160	160
A-SRTP	IN	69	70	85	90	95	110	117
	IMS	58	59	74	79	84	99	105
	OUT	63	66	81	86	91	106	115
SD-SRTP	IN	71	73	85	89	93	105	118
	IMS	59	62	74	78	82	94	108
	OUT	66	69	81	85	89	101	114
HD-SRTP	IN	71	73	85	89	93	105	116
	IMS	59	62	74	78	82	94	104
	OUT	67	69	81	85	89	101	114

Table 7: Payload size of video packets in bytes, Figure 22

Call	**Dir.**	**Min.**	**Lower whisker**	**1st Qu.**	**Median**	**3rd Qu.**	**Higher whisker**	**Max.**
SD-RTP	IN	3	1148	1148	1148	1148	1148	1151
	IMS	3	1148	1148	1148	1148	1148	1151
	OUT	3	1148	1148	1148	1148	1148	1151
SD-SRTP	IN	151	1006	1081	1106	1131	1170	1170
	IMS	129	984	1059	1084	1109	1148	1148
	OUT	139	994	1069	1094	1119	1158	1158
HD-SRTP	IN	123	1094	1122	1132	1141	1168	1168
	IMS	101	1072	1100	1110	1119	1146	1146
	OUT	111	1082	1110	1120	1129	1156	1156

Table 8: Throughput in packets per second, Figure 23

Call	Dir.	Min.	Lower whisker	1st Qu.	Median	3rd Qu.	Higher whisker	Max.
A-RTP	RX	72.00	78.00	100.00	109.75	115.50	125.00	125.00
	TX	70.00	74.00	96.50	106.25	112.50	120.50	120.50
	IMS	51.50	58.50	80.00	89.25	95.50	101.50	101.50
A-SRTP	RX	112.00	112.00	115.50	116.79	118.50	123.00	131.00
	TX	108.00	109.50	112.00	113.00	114.00	116.05	119.00
	IMS	96.00	99.50	100.00	100.50	100.50	101.01	102.00
SD-RTP	RX	196.00	218.00	260.00	277.00	288.50	307.00	307.00
	TX	194.00	222.50	256.00	273.25	281.91	297.00	297.00
	IMS	162.00	188.00	223.50	239.50	249.00	262.00	262.00
SD-SRTP	RX	165.00	165.00	236.00	268.25	345.00	504.50	504.50
	TX	163.50	163.50	226.50	263.00	322.00	423.00	466.83
	IMS	139.50	139.50	202.50	237.00	296.00	396.00	440.70
HD-SRTP	RX	209.50	209.50	327.00	516.75	585.00	812.00	812.00
	TX	205.00	205.00	322.00	504.25	581.50	691.00	691.00
	IMS	183.00	183.00	296.50	479.25	556.00	660.50	660.50

Table 9: Throughput in kilobytes per second, Figure 24

Call	Dir.	Min.	Lower whisker	1st Qu.	Median	3rd Qu.	Higher whisker	Max.
A-RTP	RX	10.18	11.59	16.77	19.03	20.50	23.33	23.33
	TX	11.56	12.66	18.34	20.60	22.23	23.71	23.71
	IMS	8.17	9.83	14.84	16.95	18.50	19.78	19.78
A-SRTP	RX	13.68	13.68	14.19	14.40	14.83	15.77	17.80
	TX	14.86	15.26	15.60	15.73	15.86	16.17	16.93
	IMS	10.99	11.33	11.58	11.70	11.77	12.02	12.02
SD-RTP	RX	114.47	163.15	179.41	185.55	190.28	200.22	200.22
	TX	118.16	167.63	183.31	189.81	193.96	204.61	204.61
	IMS	110.63	161.02	175.54	181.71	185.65	195.75	195.75
SD-SRTP	RX	50.84	50.84	126.70	161.39	246.00	396.76	396.76
	TX	53.18	53.18	122.40	158.21	224.67	346.34	399.61
	IMS	46.43	46.43	114.12	148.84	214.50	332.08	385.52
HD-SRTP	RX	101.95	101.95	232.17	447.09	527.66	712.57	712.57
	TX	104.32	104.32	234.45	440.94	530.00	582.62	582.62
	IMS	96.69	96.69	223.74	426.07	513.58	562.53	562.53

Table 10: Single call CPU load in percent, Figure 25

Call	Min.	Lower whisker	1st Qu.	Median	3rd Qu.	Higher whisker	Max.
A-RTP	0.5	0.5	1.0	1.5	1.5	2.0	2.0
A-SRTP	1.5	2.0	2.0	2.0	2.0	2.0	2.5
SD-RTP	2.5	2.5	3.0	3.5	4.0	4.0	4.0
SD-SRTP	2.5	2.5	4.0	4.5	5.5	7.5	7.5
HD-SRTP	3.0	3.0	5.0	7.5	8.5	10.5	10.5

Table 11: One-way packet delay in milliseconds, Figure 26

Dir.	Call	Min.	Lower whisker	1st Qu.	Median	3rd Qu.	Higher whisker	Max.
Audio incoming	A-RTP	0.025	0.107	0.110	0.111	0.112	0.115	0.584
	SD-RTP	0.020	0.107	0.109	0.110	0.111	0.113	0.826
	A-SRTP	0.048	0.126	0.129	0.130	0.131	0.134	0.342
	SD-SRTP	0.036	0.125	0.129	0.130	0.132	0.136	0.741
	HD-SRTP	0.021	0.122	0.128	0.130	0.132	0.138	2.283
Audio outgoing	A-RTP	0.012	0.025	0.045	0.046	0.064	0.092	0.388
	SD-RTP	0.012	0.041	0.045	0.046	0.048	0.052	0.573
	A-SRTP	0.025	0.056	0.057	0.058	0.058	0.059	0.287
	SD-SRTP	0.027	0.055	0.058	0.059	0.060	0.063	0.401
	HD-SRTP	0.017	0.049	0.058	0.059	0.064	0.073	0.456
Video incoming	SD-RTP	0.017	0.069	0.093	0.095	0.109	0.133	0.381
	SD-SRTP	0.053	0.099	0.126	0.131	0.144	0.171	1.061
	HD-SRTP	0.023	0.023	0.096	0.128	0.166	0.271	1.707
Video outgoing	SD-RTP	0.011	0.040	0.045	0.047	0.049	0.054	0.506
	SD-SRTP	0.043	0.043	0.070	0.074	0.128	0.215	0.573
	HD-SRTP	0.021	0.021	0.070	0.106	0.164	0.305	0.550

Table 12: CPU load per number of calls in percent, Figure 28

Number of calls	**Min.**	**Lower whisker**	**1st Qu.**	**Median**	**3rd Qu.**	**Higher whisker**	**Max.**
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	6.6	6.6	9.5	10.3	12.0	15.5	15.5
2	12.0	12.0	16.5	18.5	20.5	23.0	23.0
3	19.9	19.9	24.5	26.0	28.0	32.5	32.5
4	28.5	28.5	32.5	33.8	35.5	39.8	39.8
5	24.0	33.3	37.0	39.0	39.9	44.0	45.4
6	33.0	38.5	40.5	42.4	43.9	45.9	45.9
7	41.4	41.4	43.4	45.4	46.4	50.4	51.4
8	41.4	44.9	47.9	49.2	50.9	53.1	53.1
9	46.4	46.4	52.4	54.4	56.4	60.4	60.4
10	46.9	53.9	55.9	56.7	58.4	60.9	66.3

Table 13: One-way packet delay of audio packets in HD video load testing in milliseconds, Figure 31

Number of calls	**Min.**	**Lower whisker**	**1st Qu.**	**Median**	**3rd Qu.**	**Higher whisker**	**Max.**
1	0.018	0.018	0.058	0.114	0.132	0.243	1.191
2	0.016	0.016	0.059	0.113	0.130	0.236	0.860
3	0.016	0.016	0.075	0.116	0.163	0.295	2.203
4	0.016	0.016	0.073	0.116	0.175	0.327	3.965
5	0.015	0.015	0.072	0.116	0.176	0.331	6.636
6	0.015	0.015	0.073	0.116	0.179	0.338	2.814
7	0.015	0.015	0.073	0.116	0.187	0.358	2.431
8	0.015	0.015	0.073	0.116	0.191	0.368	8.166
9	0.015	0.015	0.073	0.120	0.209	0.413	4.859
10	0.015	0.015	0.071	0.119	0.207	0.410	2.691

Table 14: One-way packet delay of video packets in HD video load testing in milliseconds, Figure 32

Number of calls	**Min.**	**Lower whisker**	**1st Qu.**	**Median**	**3rd Qu.**	**Higher whisker**	**Max.**
1	0.021	0.021	0.093	0.130	0.186	0.325	2.047
2	0.018	0.018	0.107	0.131	0.198	0.334	2.340
3	0.021	0.021	0.110	0.137	0.204	0.345	11.620
4	0.021	0.021	0.117	0.159	0.239	0.422	4.771
5	0.020	0.020	0.103	0.148	0.227	0.412	9.888
6	0.017	0.017	0.104	0.156	0.259	0.491	6.495
7	0.020	0.020	0.104	0.160	0.271	0.521	6.263
8	0.018	0.018	0.105	0.165	0.303	0.600	10.973
9	0.019	0.019	0.105	0.176	0.333	0.675	37.998
10	0.017	0.017	0.102	0.168	0.317	0.639	8.194