WebRTC multipoint conferencing with recording using a Media Server

Marc Walter
Hochschule der Medien Stuttgart
Stuttgart Media University
Bachelor's Thesis

Author: Marc Walter
Walter.Marc@outlook.com
MatNr. 24650

Course: Bachelor Medieninformatik

Date: February 26, 2015

Supervisors: Prof. Walter Kriha, HdM Stuttgart
Dipl.Inf.(FH) Matthias Litz, BeamYourScreen GmbH

Name:	Walter	**Vorname:**	Marc
Matrikel-Nr:	24650	**Studiengang:**	Medieninformatik

Eidesstattliche Versicherung

Hiermit versichere ich, Marc Walter, an Eides statt, dass ich die vorliegende Bachelorarbeit mit dem Titel: **WebRTC multipoint conferencing with recording using a Media Server** selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.
Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der eidesstattlichen Versicherung und die prüfungsrechtlichen Folgen (§26 Abs. 2 Bachelor-SPO (6 Semester), § 23 Abs. 2 Bachelor-SPO (7 Semester) bzw. § 19 Abs. 2 Master-SPO der HdM) sowie die strafrechtlichen Folgen (gem. § 156 StGB) einer unrichtigen oder unvollständigen eidesstattlichen Versicherung zur Kenntnis genommen.

Stuttgart, den 23.02.2015

Marc Walter

Abstract

Goal is to both research and implement a multipoint communication solution that uses WebRTC for audio and video transmission. The conference size should be arbitrary but support at least four participants and allow multiple conferences at the same time. The solution should contain recording functionality and allow participation of devices in restrictive network environments.

These goals were met by using a a media server that allows to significantly increase the conference size from the maximum of four when using a full mesh peer-to-peer architecture.

This thesis is separated into two parts: The first consists of an introduction to WebRTC, research and theoretical assumptions that lead to an implementation strategy. The second part contains information about the selected media server, follows the implementation of the prototype solution and describes parts of its architecture.
The prototype itself is not part of this thesis.

Included are benchmark results for WebRTC conferences concerning the bandwidth requirements for different multipoint architecture patterns and screen resolutions. Those were conducted using the prototype application.

Zusammenfassung

Ausgangspunkt für diese Thesis ist das Ziel, eine auf WebRTC basierende Webanwendung als Kommunikationslösung zu schaffen, die mindestens vier Teilnehmer in einer Konferenz kommunizieren lässt, mehrere Konferenzen gleichzeitig unterstützt, die Aufnahme der Konferenzen ermöglicht und bei der auch Teilnehmer in restriktiven Netzwerkumgebungen teilnehmen können. Dieses Ziel wurde in vollem Umfang erreicht.

Hierbei wird ein Media Server eingesetzt, um die maximale Teilnehmeranzahl bei Konferenzen im Vergleich zu direkten Peer-To-Peer Verbindungen, die bei vier Teilnehmern je nach Verbindungsgeschwindigkeit an ihre Grenzen stoßen, fast beliebig zu vergrößern.

Nach einer Einführung in WebRTC und Vorüberlegungen für die gewünschte Kommunikationslösung im ersten Teil dieser Thesis werden im zweiten Teil die Auswahl eines Media Servers sowie der Prototyp und dessen Architektur beschrieben.
Diese Prototypanwendung ist jedoch nicht Teil dieser Thesis.

Des Weiteren wurden mit der Prototyp-Anwendung Vergleiche zu Bandbreitenanforderungen von WebRTC bei der Verwendung von verschiedenen Kommunikationsarchitekturen und Bildgrößen der Videoaufnahmen durchgeführt und dokumentiert.

Contents

Part I

Theoretical part

Chapter 1

Introduction

Basis of this thesis was the plan to create a multi party communication solution using the peer-to-peer technology WebRTC.

This first part contains the definition and reasoning for the goals of the solution, an introduction to WebRTC, benefits of using a media server and a section about different components needed to achieve the set goals and create a working prototype.

The second part starting on page 31 contains information about the implementation of the simucos prototype.

1.1 Definition of goals

Before starting on the rationale for the prototype, the goals need to be defined:

Goal is a WebRTC conferencing prototype that

1. allows more than four participants to communicate simultaneously
2. supports recording of conversations
3. allows participants in restrictive network environments to take part in conversations
4. optionally allows SIP-Clients (soft/hard phones), and Teleconferencing systems to connect to a conference

1.2 Introduction WebRTC

Web Real-Time Communication (WebRTC) is an effort that was started in 2011[1] to enable direct real-time communication between two browsers without needing to install any browser plug-ins or platform-specific applications.

Neither does WebRTC place any restrictions on the type of devices like computers, mobile phones or TVs.

Usually applications in a browser communicate with a web server to receive and send data, but when using WebRTC a direct communication channel between two peers is opened without using any servers for the transmission of data.
The signaling that is needed for two browsers to communicate with each other is usually done using a web server, forming the WebRTC triangle as depicted in figure 1.1.

In order for WebRTC browsers to connect to another peer, signaling is needed. This is achieved using an SDP offer and answer mechanism, allowing the peers to exchange IP addresses, port numbers and other information like supported codecs.

This signaling is not defined in WebRTC and usually a web server is used to help peers discover each other and to exchange the SDP information, forming the mentioned triangle. After the initial exchange, the web server is not needed anymore.

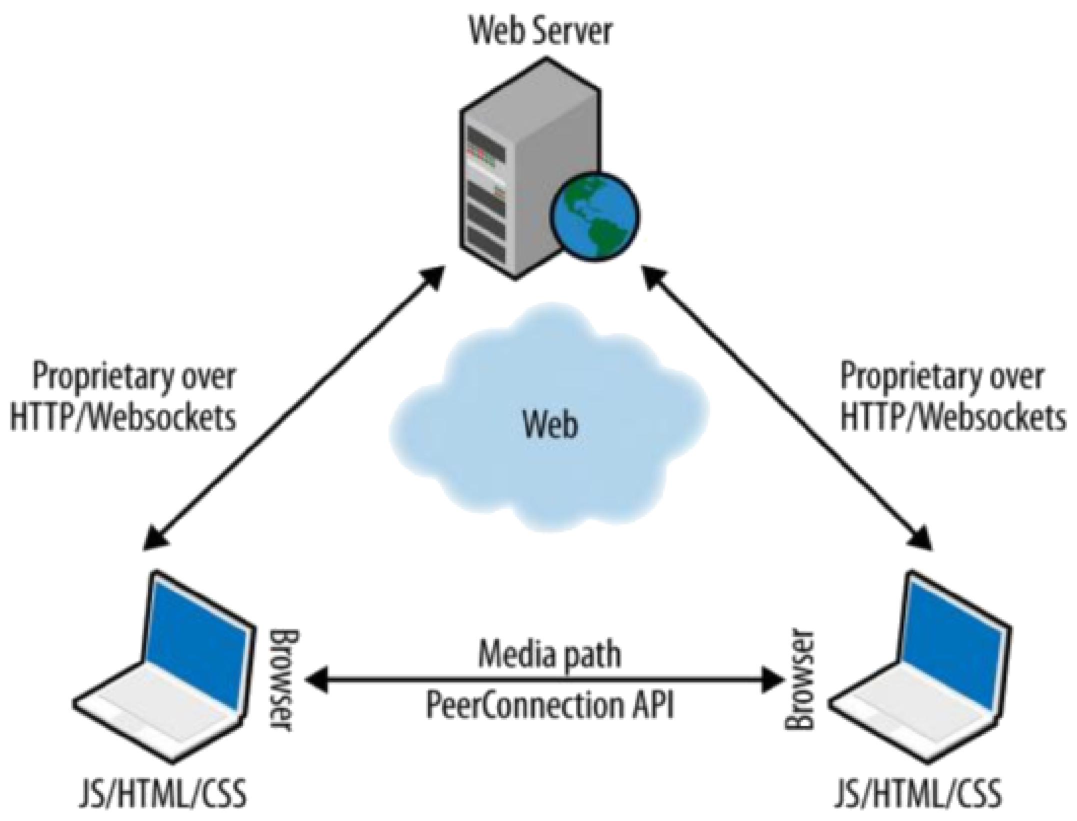


Figure 1.1: WebRTC triangle[2, chapter 1]

1.2.1 What does WebRTC do

WebRTC exposes ECMAScript[1] APIs that allow a participant to use a web application that retrieves audio and video streams from cameras or microphones.

Also it allows to establish a peer-to-peer connection using standardized protocols between two compatible browsers. This connection may be used to send arbitrary data in both directions between the two peers, for instance a real-time video chat..

1.2.2 ECMAScript API

The API exposed by the browser contains three major interfaces:

The **RTCPeerConnection** object is the main object used by a web application in the browser. It encapsulates the creation of connections between peers[3], handles and parses SDP[2]offers and answers, has an associated ICE agent to find and negotiate usable IP addresses and port numbers, and both receives remote media streams and sends local media streams using the peer connection.

A media stream is requested in the browser using the **MediaStream API**[4, section 10.2.1] and can afterwards be used in the RTCPeerConnection.

Arbitrary non-media data is sent between peers using the TCP based **RTCDataChannel**[5]. It is used to send files, information about the quality of the stream or arbitrary messages.

[1]JavaScript is one implementation of [ECMAScript] as defined by ECMA-262

[2]SDP contains information how to initialize multimedia communication sessions, like IP addresses, media types or media codecs and was defined in [RFC 4566]

1.2.3 Types of WebRTC-compatible devices

The current standard defines different types of WebRTC-compatible devices[6, section 2.2].
The most important ones are:

First a **WebRTC device**, it conforms to the protocol specifications, meaning it supports the needed UDP based protocols for the peer connection and the TCP based ones for the the data channel, including encryption for both packet types using TLS and DTLS.

A **WebRTC browser** is the second type, which is a WebRTC device that also supports the full ECMAScript API.

The third is a **WebRTC gateway**, which is a WebRTC device that mediates media traffic to non-WebRTC devices and may not conform to all protocol specifications.

A **WebRTC gateway** might be used to enable audio-only communication between a PSTN telephone and a WebRTC browser or one-way communication between a participant using a WebRTC browser and another device that only shows the first participants stream, but cannot send anything on its own.

1.2.4 Is it ready yet?

In February 2015, three browsers (Google Chrome, Mozilla Firefox and Opera) support real-time communication with WebRTC directly, and the mobile platforms Android and iOS are also supported.

That way 60-80% of desktop browsers[3]are compatible, and also about 90%[4]of all smart phones could use WebRTC.

Additionally, browser plug-ins exist for both Internet Explorer and Safari to support WebRTC[5]. This increases the number of potentially WebRTC compatible desktop browsers to about 90%.

Microsoft is also planning to support WebRTC in its coming browser and is actively contributing to the ORTC[8] specification[9] and its implementation, which is based on the WebRTC 1.0 specification[10] with some minor changes[6]and maintained by the *W3C Object-RTC API Community Group*.

1.2.5 More information

Currently, two standards organizations collaborate to develop WebRTC:

The browser API[10] is defined by the *W3C WebRTC Working Group* and contains the expected behavior and functions a WebRTC compatible browser needs to implement.

The *IETF RTCWEB Work Group* defines the protocols used for the transmission of content, how connection management is initialized and security considerations[6].

[5]Statistic gathered by StatCounter
`http://gs.statcounter.com/#desktop-browser-ww-monthly-201408-201501-bar`

[5]This statistic was gathered by IDC[7]

[5]One of those is developed by temasys `https://www.temasys.com.sg/solution/webrtc-plugin`

[6]Differences between WebRTC and ORTC include exchanging the text-based SDP for a JavaScript Object Model, and access to lower level functions of the media stream transmission.[9].

Chapter 2

Goals to achieve with the solution

2.1 Multipoint conferencing with WebRTC

To achieve multipoint communication, several architectures have proven usable.

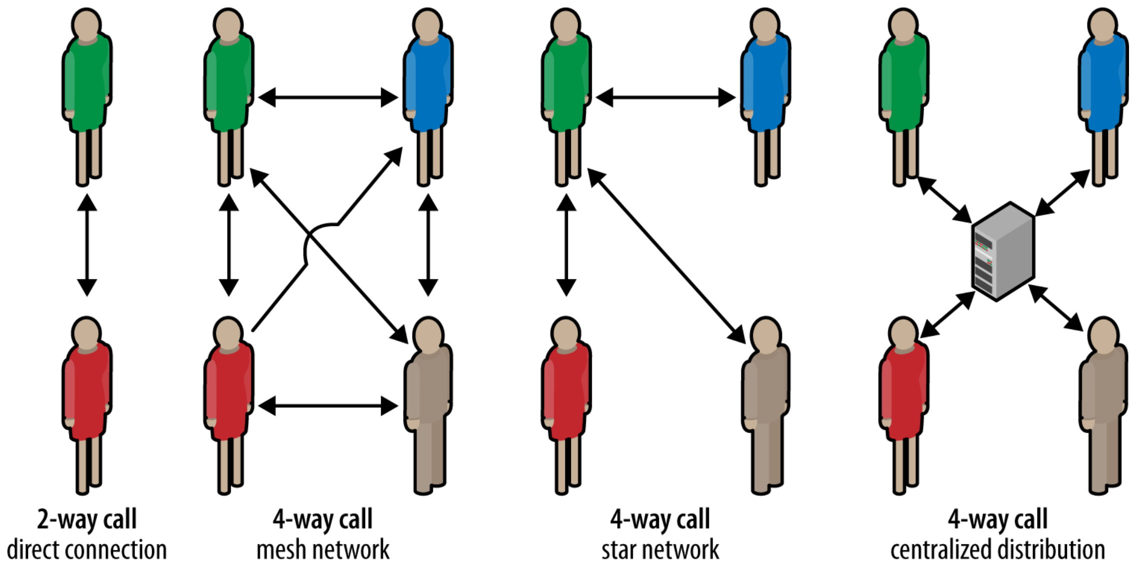


Figure 2.1: Possible multipoint communication architectures
Source: Ilya Grigorik[11]

As WebRTC is a technology that is used to create direct peer-to-peer connections between two entities, the most natural way of connecting multiple participants would be to use a full mesh network as displayed in figure 2.1 for a four-way call.

The figure also shows that the number of connections in a full mesh network increases exponentially with the number of participants, which limits the size of a reliably working conversation to about four[1].

Apart from the bandwidth the second limiting factor to a conversation size is the processing power of the used device.

While most modern mobile devices have a hardware decoder for H.264, the support for VP8 is on most devices limited to the software which adds more processing strain.

In a conference with four participants, the device needs to encode one outgoing media stream and upload it to three different destinations, and also decode three incoming video streams in real time or having a conversation will not work reliably.

[1]The tests conducted with the prototype implementation in section 8 on page 39 showed that connecting more than four participants in a full mesh network would consume about 6.1 Mbit/s of bandwidth at a resolution of 640x480 pixels and 15 frames per second. The bandwidth needed for a conversation with seven participants would exceed the maximum possible bandwidth in a WLAN 802.11b network.

2.2 Recording conferences

In order to fulfill the goal of recording conferences two questions need consideration first: Where a recording is created, and how a recording is created.

2.2.1 Where to record conferences

To record a conference, three possible setup variants are possible, that do not change the specifics of the record mechanism.

A user records the conference

The simplest option would be to record a conversation on the client side. As each client already receives every participant's media stream in order to display it, the same videos that are displayed to the client might be recorded to the client's file system.

Unfortunately, this would need a high amount of processing power on the client's side, that just might not exist on a mobile handset device or slow down another crucial application running on the clients computer.

Also, it would complicate the distribution of the recorded conference, because each participant would need to request the specific file from the recording client or record the conference himself.

A dedicated client records the conference

A client connects to the conference without sending any data, but only receives the data sent by other participants.

This would limit the amount of processing power needed on the recording device as it does not need to record, encode and transmit its own media stream, but only receives streams.

Unfortunately it would also increase the number of connections needed to make the conference work as it increases the participant count by one.

This client could be either another device one of the participants owns, or it might be a computer program on a server, which would also be able to distribute the recorded conference after it was ended.

A central entity records the conference

If each participant would send his stream to a central entity, that entity might be used to record the conference.

The central entity could act like another participant, which would be just like option two, or it could be used to implement a centralized architecture.

Two different types of devices may be used for this:

The first would be a **TURN server** because it might already be needed for some networks to create a connection[2], but it needs to be able to break the TLS encryption of the communication in order to decode the media stream and save it.

Or a **media server** might be used that additionally needs to support the WebRTC protocols and the two mandatory codecs VP8 and MP4.

[2]A TURN server is needed in some NAT environments, as described in section 2.3 on page 11.

2.2.2 Record mechanism

How to record a conference boils down to two possible actions:

First, each participant's stream is saved directly to a file. This might also be done on a clients side, and the recorded file could be uploaded to a media server or just distributed to all other participants afterwards.

The second option would be to record all participants in the conversation. In that case still each participant might be recorded separately and mixed together into one video file. Either as multiple tracks in one video[3], or mixed into one video that shows for instance all videos next to each other like a video wall or another setup that shows for instance the current speaker and maybe the last few speakers, too. Some examples are depicted in figure 2.2

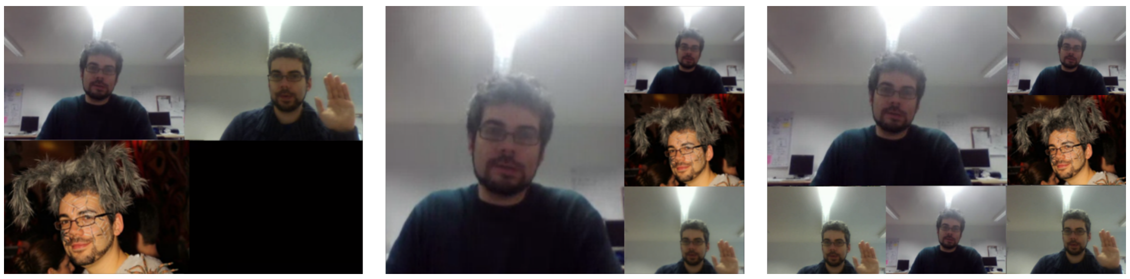


Figure 2.2: Exemplary video wall configurations

If a custom video player tool is used, the information which participant is speaking at the moment might be saved and the tool can use either multiple files or the file with multiple streams to display the most active/important speaker correctly.

[3]For example the matroska file format[12] allows multiple tracks.

2.3 Enabling a connection in a NAT environment

2.3.1 Network Address Translation (NAT)

NAT is a mechanism that is in wide adoption both because the number of IP addresses is finite and due to security concerns: The routing device translates IP addresses from one space to another and is both able to masquerade the internal IP addresses, and to provide a convenient way to send data to the desired host.

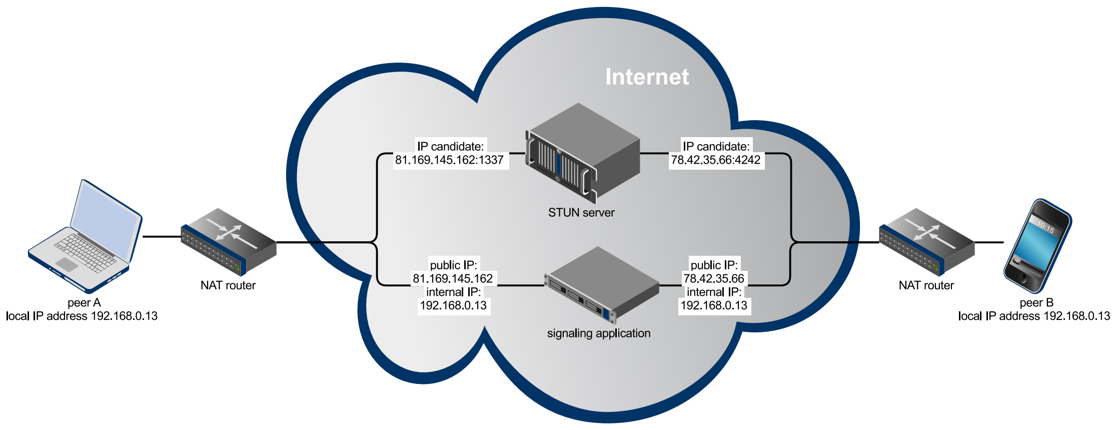

Internet
IP candidate: 81.169.145.162:1337
IP candidate: 78.42.35.66:4242
STUN server
public IP: 81.169.145.162 internal IP: 192.168.0.13
public IP: 78.42.35.66 internal IP: 192.168.0.13
signaling application
NAT router
NAT router
peer A local IP address 192.168.0.13
peer B local IP address 192.168.0.13

Figure 2.3: Network Address Translation[4]

If a direct connection between peer A and peer B should be started, it will not work if only a signaling application is used, because the signaling application only knows the public IP address, which belongs to a routing device.

Even if the peer were to pass his local IP address to the signaling application, still no connection could be created because the clients' IP address would not lead to the client in the signaling application's address space, or like in figure 2.3 both devices could even propagate the same local IP address.

Masquerading an IP address behind the router's IP address and a specific port number is often call port address translation (PAT) or network address and port translation (NAPT) and very common, because the number of IP addresses is finite. The router uses the port number on incoming connections to reroute the packet to the specific host without allowing any direct connections to this host or publicly announcing its IP address.[13]

In order to establish a two-way connection the signaling application needs to gain access to a reliable IP address and port number combination, which will arrive at the desired peer.

2.3.2 Interactive Connectivity Establishment (ICE)

To achieve this, the Interactive Connectivity Establishment technique was standardized in [RFC 5245], and provides a protocol to enable NAT traversal for UDP-based multimedia sessions with an offer/answer model.

It consists of two sub protocols: The **Session Traversal Utilities for NAT (STUN)** protocol and in case the **STUN** protocol does not return usable IP candidates, **Traversal Using Relays around NAT (TURN)**, which is actually an extension to **STUN** may be used.
Both protocols communicate with a publicly available server.

[3]The diagram was created with the web tool gliffy `http://www.gliffy.com/`

STUN

The STUN protocol uses the server to open connections from the peer using UDP packets.
First it sends an offer and the server responds with an answer containing the public IP address of the client and a port number.

This process is repeated several times to ensure all possible IP candidates are gathered and the client may try multiple combinations before not being able to establish a connection.

For most cases[4], STUN is enough to establish a connection, but sometimes the NAT router is too restrictive and a connection won't be able to created.

Problems finding public IP candidates using STUN

An example for such a setup is a symmetric NAT[RFC 3489, section 5], where every time when a client tries to access another server, another port is used for the mapping.

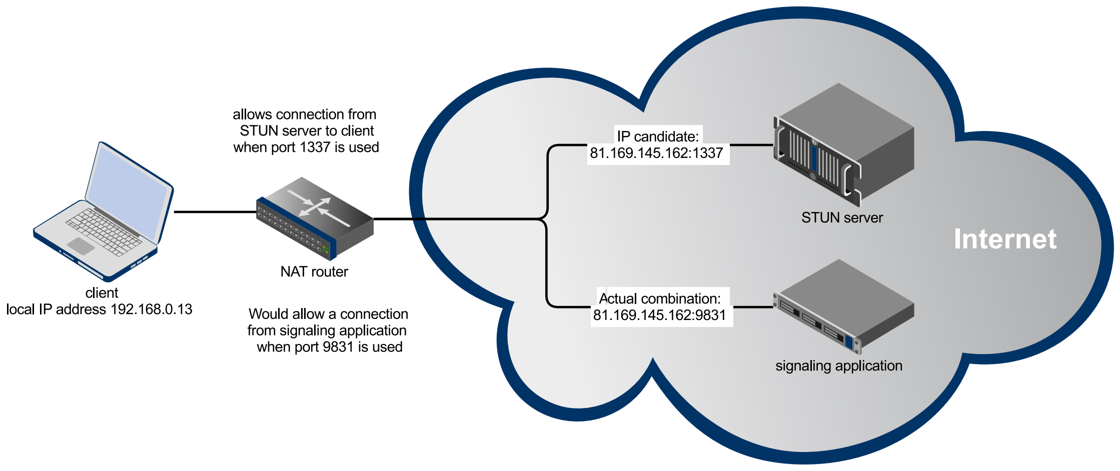

allows connection from STUN server to client when port 1337 is used
client
local IP address 192.168.0.13
NAT router
Would allow a connection from signaling application when port 9831 is used
IP candidate: 81.169.145.162:1337
STUN server
Internet
Actual combination: 81.169.145.162:9831
signaling application

Figure 2.4: Example for a symmetric NAT[4]

In this case, the STUN server will return an IP candidate that can establish a connection to the STUN server, but if the client uses this candidate to open a connection to another server, that server's answer will be rejected by the router because it would need to use the port assigned to the new connection and not the one to the STUN server.

TURN

To create a two-way connection in such an environment, all data needs to be routed over another relay server. This is achieved using the TURN protocol and a TURN server.

Figure 2.5 shows how a connection between the TURN client and Peer A or Peer B is established using the TURN server as a connector, relaying all data over it.

That way it is possible to create a connection to the peers, but the TURN server adds more delay to the transmission and has high maintenance costs.

Using TURN servers should generally be kept to a minimum.

[4]In 2005, a huge number of networks were tested by ISPs and Bryan Ford[14], and 82% of them worked with STUN

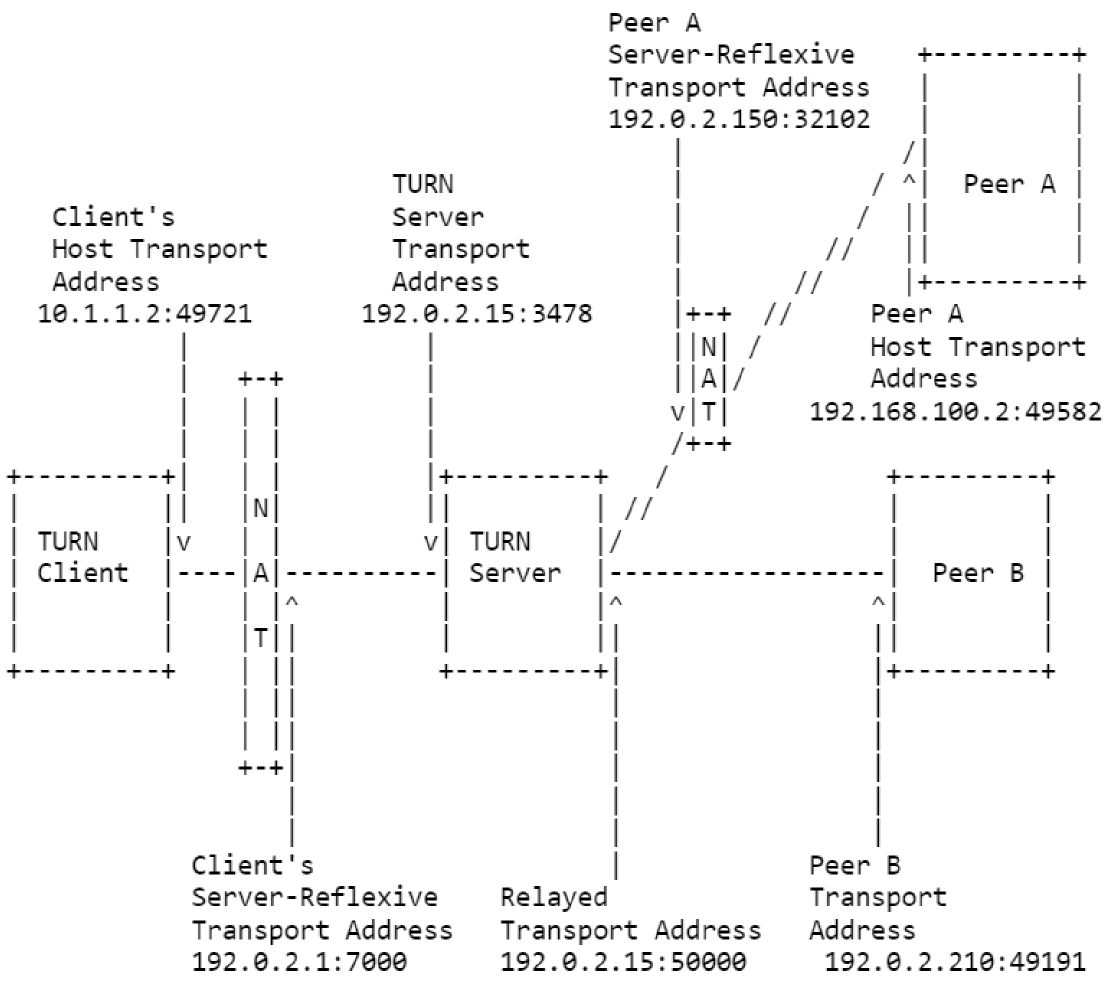

Peer A Server-Reflexive Transport Address 192.0.2.150:32102
Peer A
TURN Server Transport Address 192.0.2.15:3478
Client's Host Transport Address 10.1.1.2:49721
NAT
Peer A Host Transport Address 192.168.100.2:49582
TURN Client
NAT
TURN Server
Peer B
Client's Server-Reflexive Transport Address 192.0.2.1:7000
Relayed Transport Address 192.0.2.15:50000
Peer B Transport Address 192.0.2.210:49191

Figure 2.5: Traversal Using Relays Around NAT (TURN)
Source: [RFC 5766]

2.3.3 Specifics about the ICE protocol in WebRTC

WebRTC uses the gathered IP and port candidates to establish a direct connection to the other peer.

First the peer connection object tries to connect to all candidates using UDP, if that fails it tries to open a connection using TCP, then http and in the end https[11, chapter 18].

If all of these tries do not result in an opened connection, a TURN server will be used if one was defined or communication between the two peers will not be possible.

2.4 Connecting SIP compatible devices with WebRTC devices

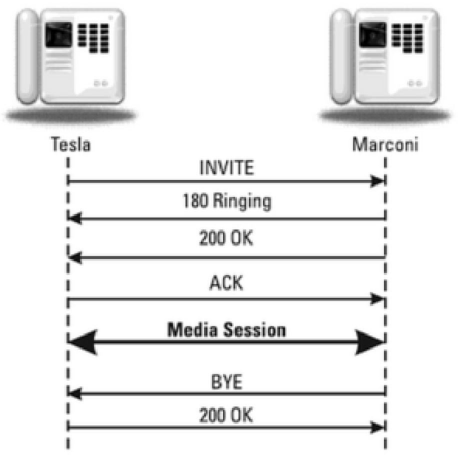
2.4.1 About the Session Instantiation Protocol (SIP)

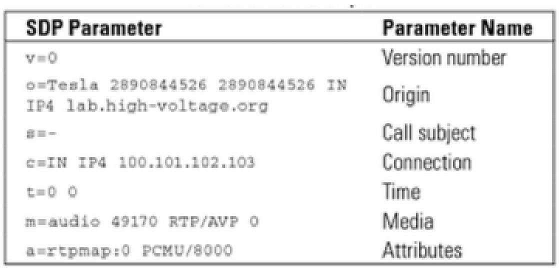
SIP is a standard to facilitate Voice over IP or video chat communication sessions defined in [RFC 3261].

SIP clients exist for instance as Software or hardware IP telephones. Also many Teleconferencing systems use the SIP protocol. Each device or client is called a User Agent (UA).

Figure 2.6 shows how a connection is established by first sending an `INVITE` message to the other party. This message also contains the SDP information as defined in [RFC 4566] that could look like depicted in the same figure. This information is used by the called UA to select the communication channel and codecs to be used. Then he returns an `OK` message and after the caller has sent an acknowledgment, the media session is established.

The codecs are passed as a sorted-by-preference-list, so the callee can choose the codec that fits both UAs best.





SDP Parameter	Parameter Name
v=0	Version number
o=Tesla 2890844526 2890844526 IN IP4 lab.high-voltage.org	Origin
s=-	Call subject
c=IN IP4 100.101.102.103	Connection
t=0 0	Time
m=audio 49170 RTP/AVP 0	Media
a=rtpmap:0 PCMU/8000	Attributes

Figure 2.6: SIP session establishment example with excerpt of SDP data
Source: [15, chapter 2]

2.4.2 Problems when connecting SIP and WebRTC devices

Codecs

Mandatory codecs of WebRTC[5] are Opus as defined in [RFC 6716] and G.711 in PCMA and PCMU as defined in [RFC 3551, section 4.5.14].

There are no restriction on the codec that a SIP device may use, but also there is no mandatory codec that all devices must understand. This may cause problems when WebRTC and SIP devices try to communicate.

Types of devices

SIP does not enforce specific requirements on its User Agent devices, for instance a PSTN telephone would be treated equally to a teleconferencing solution that supports five different cameras and as many microphones.

[5]Mandatory Codecs for WebRTC are defined by the IETF[16]

2.4.3 How to connect SIP and WebRTC

To include SIP compatible devices into a WebRTC conference, either the SIP UA needs to support one of the mandatory WebRTC codecs, or a media bridge is needed to transcode the streams to formats that the endpoints understand.

A WebRTC gateway could be used for instance to convert the H.323 standard to an Opus or G.711 audio stream and connect a PSTN telephone to the WebRTC session.

This media bridge functionality could also be provided by a central dedicated media server, where direct support for any needed codecs could be installed.

Chapter 3

Benefits of using a central media server

3.1 Concerning the goals

As the title of the thesis already suggests, a media server was chosen to work as a central hub in the setup.

This decision simplifies reaching the goals defined in section 1.1 and results in some additional side benefits like additional architecture options[1].

3.1.1 Recording

Even though for recording no central media server is needed, each participant's stream would have to be sent to the recording entity, adding to the total number of connections.

Also, because not every participating device might have the necessary processing power and free space in its file system to record the conversation, no participant should be chosen as a recorder without their consent.

Adding recording to the media server is simple because the connection between the peer and the media server is already decrypted on the server and the encryption does not need to be broken as it would be the case when a TURN server would be used.

3.1.2 Number of participants

As each participant sends one stream to the recorder, it would also be a good idea to use the recorder as a central multicast point to reduce the number of needed connections.

Even if each participant were to receive the unaltered stream of every other participant, at least the number of outgoing streams would be reduced to one on the clients' side.

The media server, with a faster connection and higher processing power than any of the participants, would handle the multicasting of the stream and distribute it to all other participants.

Each media stream is decrypted end-to-end, which is why a simple hub would not suffice in order to reduce the strain on the clients' side, because to multiply the stream, the distributing hub needs to be able to access the decrypted data streams.

Just like the recorder, the multiplexer would need to break the encryption. When using a media server this is not necessary as each peer connection ends at a WebRTC endpoint on the media server anyway.

3.1.3 Enabling a connection in a NAT environment

Ensuring participants behind a NAT to connect is easier when a Media Server is used, because the Media Server itself will have a publicly available IP address and connecting to it will not be a problem for any participant.

In order to retrieve a public IP and port combination, a STUN server will be needed. A great option would also be to use the media server as the STUN server, as even if the participant is located behind a symmetric NAT[2], the connection would be established.

[1]Relayed and mixed architecture as described in section 3.2.

That way the number of networks where a TURN server is needed to establish a connection with the media server would be further decreased[3]

3.1.4 Connecting to SIP networks/devices

In order to connect SIP compatible devices to a conversation on the media server, a WebRTC gateway could be used that will connect to a WebRTC endpoint on the media server.
Also ensuring a connection between a telephone switching system[4]and one WebRTC gateway or media server lends fewer obstacles than connecting the telephone switching system to each WebRTC compatible device.

Another benefit of using a Media Server is, that it can act as a media bridge and transcode the streams between proprietary or open codecs to the supported WebRTC codecs and allow conferences with otherwise not compatible SIP clients or even the inclusion of PSTN telephones.

For instance trancoding between the widely used ITU codecs[5]that are available in H.323 or SIP environments[17], but are not supported by WebRTC compatible browsers as of now, or any other number of proprietary audio or multimedia codecs.

[3]The number of networks needing a TURN server would be lower than the 18% that Bryan Ford[14] measured.

[3]A NAT where each connection to another server is routed through another IP and port combination, as depicted in figure 2.4

[4]For example Asterisk `http://www.asterisk.org/`, an open source telephone PBX (private branch exchange) system

[5]Several audio codecs are specified by the ITU[18], for instance G.722 and G.729

3.2 Possible multipoint conferencing architectures

All multipoint conferencing architectures in this section are based on a generic star structure, as depicted in figure 3.1 and use the media server as a central hub.

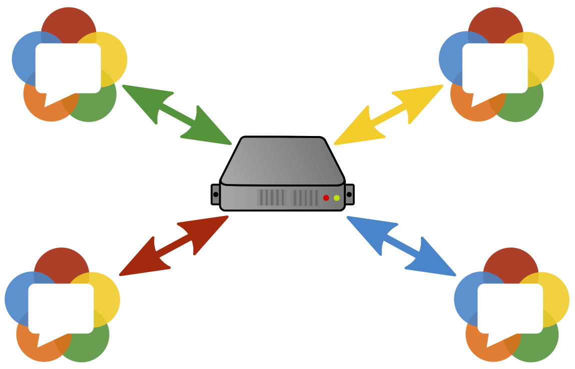


Figure 3.1: Star network

3.2.1 Full mesh network

Without a central hub, the only option to connect all participants would be using a full mesh.

It would also be possible to use one participant as a central point but that would add too much strain on the device for conferences with many participants.

In a full mesh, each participant sustains one full-duplex connection to every other participant as shown in figure 3.2[6].

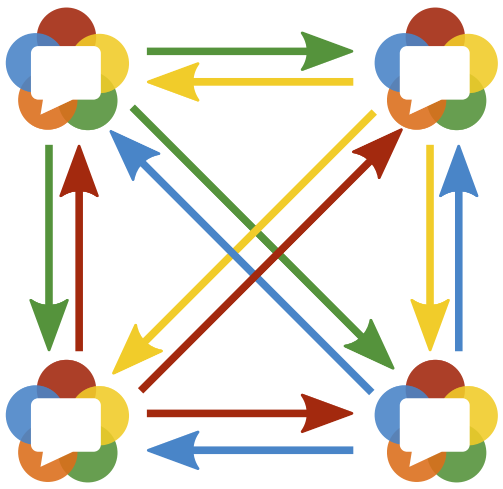


Figure 3.2: full mesh network with four participants

[6]This limits the maximum feasible conference size as shown in section 8 on page 39.

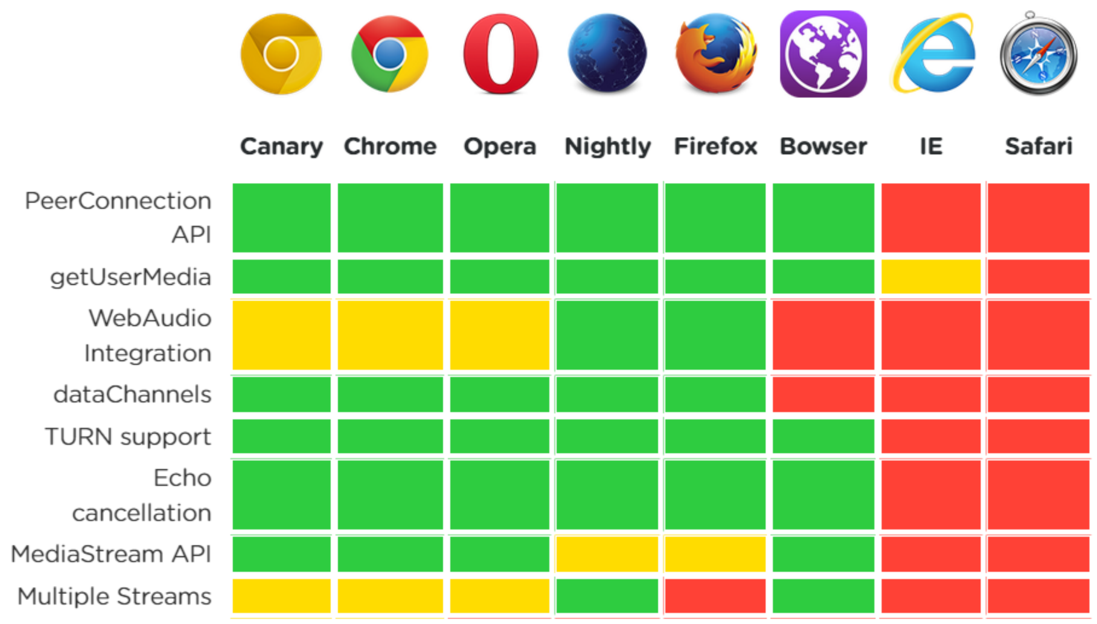
3.2.2 Clarifications

About media streams and media tracks in WebRTC

The WebRTC standard allows one media stream to contain multiple audio and video tracks.[4, section 4]

This would allow the media server to add multiple participants to one file stream without needing to recode the entire stream.

Unfortunately, this feature is not usable for this solution yet. Even though both Google Chrome and Mozilla Firefox (nightly build) support multiple tracks, the implementation and interaction differs too much to be compatible.
Also Mozilla Firefox has trouble adding tracks to an existing media stream.
The current state of support for multiple media tracks in one stream as of February 24, 2015 is depicted in figure 3.3, where the feature is called *Multiple Streams*.



	Canary	Chrome	Opera	Nightly	Firefox	Bowser	IE	Safari
PeerConnection API	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
getUserMedia	🟩	🟩	🟩	🟩	🟩	🟩	🟨	🟥
WebAudio Integration	🟨	🟨	🟨	🟩	🟩	🟥	🟥	🟥
dataChannels	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟥
TURN support	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
Echo cancellation	🟩	🟩	🟩	🟩	🟩	🟩	🟥	🟥
MediaStream API	🟩	🟩	🟩	🟨	🟨	🟩	🟥	🟥
Multiple Streams	🟨	🟨	🟨	🟩	🟥	🟩	🟥	🟥

Figure 3.3: Current multiple track support in a media stream
Source: &yet[19]

3.2.3 Architecture definition

Three properties can be used to define the different types of media servers and their network architecture. Since multiple sources were used to gather this list, and most sources use different naming schemes or meanings for specific words, this part will first explain what each property means in this document[7].

Stream transmission

A media server transmits streams either in a **specific** or a **general** manner:
In a **specific** transmission mode, each participant receives one or more streams tailored for that participant, he would never receive the video stream he sent to the media server. The media server

[7] Naming is kept close to the naming used in the IETF document about RTP topologies defined in [RFC 5117] and the draft of the obsoleting document for [RFC 5117], in its current state[20].

needs to either duplicate or transcode all incoming streams and create a new stream for each participant, resulting in a high workload.
When all participants receive the same stream(s) from the media server, this is called a **general** transmission. It is possible that a participant receives the same stream that he already sent to the media server.

Stream handling on the media server

Relay or Mixer: A **relaying** media server sends multiple media streams to each participant. Also a media server that uses one media stream to send multiple media tracks to each participant is considered a **relay**.
A **mixing** media server sends only one media stream to each participant. If it sends only the video of one participant or mixes multiple streams into one new video is not relevant.

Selection of streams

A media server could act either as a **Selective Forwarding Unit (SFU)** or a **Multipoint Control Unit (MCU)**: While a **MCU** would ensure that every participant receives the stream of every other participant, a **SFU** would only forward a selected number of streams.

For instance could a **SFU** be configured to only forward the streams of the currently speaking participants and to drop the other participants' streams.
Another Option would be to display four participants in a grid and only forward the audio streams of the additional participants.
This would limit the transferred data to what is actually needed, and as only one mixed video is sent to all participants, the media server only needs to encode this one stream resulting in lower server costs.

A **MCU** would ensure that each participant receives all data streams, but would have high requirements concerning the network interface, which would result in high server costs.

3.2.4 Examples

How would a certain type of media server behave?
Some examples:

1. **A relayed specific MCU (RSM)**
   The media server receives the media streams of each participant and sends it to every other participant. Before sending the stream to a participant the stream is transcoded to another resolution depending on the recipient's screen size.

2. **A mixed general SFU (MGS)**
   The media server receives the media streams of each participant and transcodes them as necessary. It sends the same video to every participant, but the video shows a grid with the video feeds of the last nine speakers and contains all mixed audio channels.

3. **A relayed general SFU (RGS)**
   The media server receives the media streams of each participant and transcodes them as necessary. It sends the same four video streams to every participant, but each as its own media stream. Three of the streams show the first three participants in the conference, the fourth shows the currently speaking participant.

3.2.5 Comparison of multipoint architectures

Whether a media server is used as a MCU or a SFU does not affect the following comparison much, thus it is omitted for brevity.

Relay

A **relaying** media server sends multiple media streams to each participant. Also a media server that uses one media stream to send multiple media tracks to each participant is considered a **relay**[8].

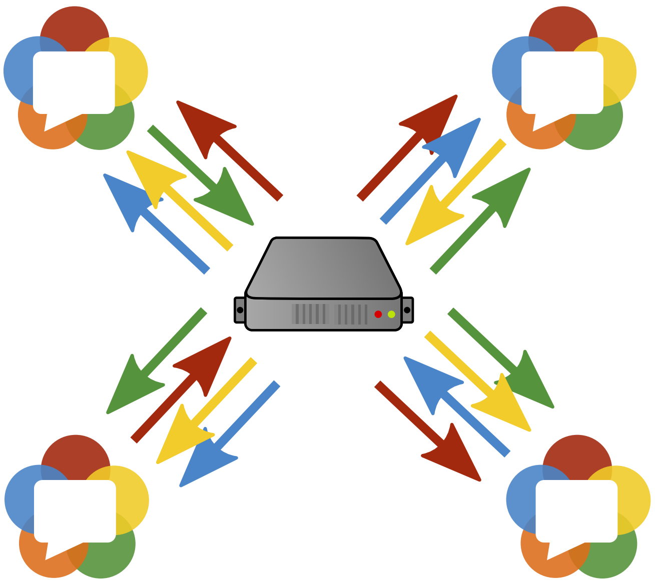


Figure 3.4: A relaying media server

The media server is used as a relay and sends multiple streams to each participant. The needed processing power on the server is low, but the bandwidth requirements are high.

For a participant, a relay is similar to a full mesh network between all peers as he receives one stream per other peer in the network. But each participant only needs to upload one stream, resulting in a lower needed bandwidth on the client's side.

[8]Instead of relay, very often the term router is used

Table 3.1: Comparison relaying media server

Relayed Specific MCU/SFU		
	Pros	**Cons**
participant	handle two media streams	decode $n-1$ video tracks and ($n-1$ or 1) audio tracks Mobile phone might receive an UHD stream
media server	no transcoding[9]	handle many streams (n incoming and $(n-1)\cdot n$ outgoing)

Relayed Specific MCU/SFU		
	Pros	**Cons**
participant	handle two media streams Media tracks are generated for the participant (network settings, screen size)	decode $n-1$ video tracks and ($n-1$ or 1) audio tracks
media server		handle many streams (n incoming and $(n-1)\cdot n$ outgoing) transcode many streams[10] detect participant performance indicator[11]

[9]The media server would need to transcode between codecs if the participants do not support the same codec. But in WebRTC both H.264 and VP8 are mandatory for the participants' devices to implement.

[10]Either for each participant or for a number of presets

[11]network speed, screen size

Mixer

A **mixing** media server sends only one media stream to each participant. If it sends only the video of one participant or mixes multiple streams into one new video is not relevant.

A media server that uses one media stream to send multiple media tracks to each participant is considered a **relay**.

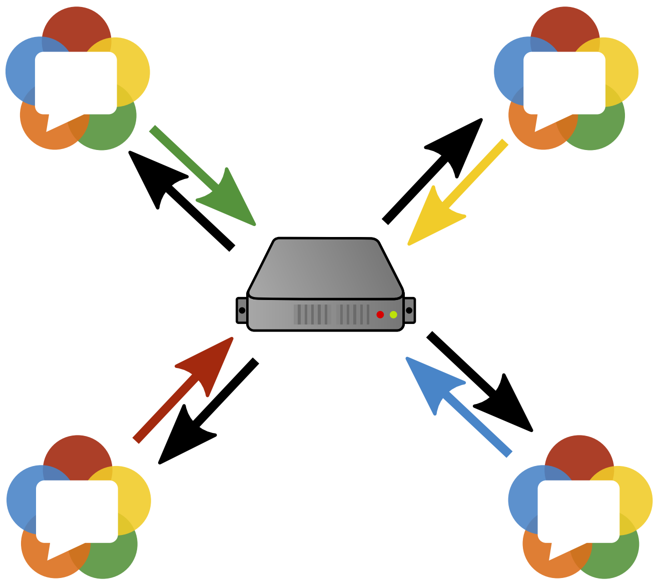


Figure 3.5: A mixing media server

The media server receives all participants' media streams and sends one media stream in return. The needed processing power on the server is very high if a new video stream is created, but the bandwidth requirements are low.

For a participant, a mixer is similar to a direct peer-to-peer connection. Each participant uploads one stream and downloads one stream. The bandwidth requirement on the client's side remains the same no matter how many participants attend the conversation.

Table 3.2: Comparison mixing media server

Mixed Specific MCU/SFU		
	Pros	**Cons**
participant	handle two media streams	Participant might receive himself
	decode 1 video track and 1 audio track	Mobile phone might receive an UHD stream
media server	stream one video to all participants	mix all incoming video streams into new one
	number of connections (n incoming and n outgoing)	One codec must be available on all devices

Mixed Specific MCU/SFU		
	Pros	**Cons**
participant	handle two media streams	cannot record all participants of conference locally
	decode 1 video track and 1 audio track	
	Media tracks are generated for the participant (network settings, screen size)	
media server	number of connections (n incoming and n outgoing)	mix all incoming video streams into new one
		transcode many streams[12]
		detect participant performance indicator[13]

[12]Either for each participant or for a number of presets
[13]network speed, screen size

Chapter 4

Components for the solution

In order to achieve the goal of a web application that allows multiple participants to start and record conversations using WebRTC three components are needed:
A media server, a signaling application and a STUN or TURN server.

For media processing and handling a dedicated **media server** should be used, because the demands to processing power and network interface throughput are very high.

The **signaling application** is the linking element between the participants in a conversation, and between the media server and the participants. It needs to create rooms on the media server and send participants' data to it. It may also be used as a load balancer to distribute participants to multiple media server instances.

A **STUN or TURN server** is needed to provide the participants with their public IP addresses and ports if they are located behind a NAT router.

4.1 Media server

The main purpose of media servers is the handling of many media streams while increasing the delay of the stream as little as possible.

It needs to transcode media streams between different resolutions and codecs as fast as possible. Transcoding of media streams in real time is needed if it should be used in a communication application.

And it allows multiplexing of one stream, a resources-saving way of splitting or multiplying a media stream and sending it via its network interface to multiple recipients.

4.1.1 Hardware or software based

Both hardware and software based media servers exist, but for this document only software based media servers are relevant, because specific hardware is very expensive and not easily obtainable.

A **hardware based** media server would for instance use a customized processor architecture and an adjusted operating system[1]. In the last years, many graphics cards were added to a server to allow faster video encoding and decoding, but the algorithms are often not optimized for the massive parallelization that is possible when programming graphic cards.
Also, hardware cards exist for encoding for instance H.264 video streams faster than would be possible using multiple CPU cores.

A **software based** media server uses multiple CPU cores to achieve parallelization and the media server programs are usually written in C and optimized for speed and only media handling.

[1]For instance a RISC based one like the HP 9000 series[21].

4.1.2 For the purpose of a WebRTC media server

In this case, the media server needs to be able to act as a WebRTC device and create a peer connection to a WebRTC browser.
It needs to negotiate and establish the encrypted connection to the other peer and receive and send the media stream containing both audio and video tracks.

The ability to receive arbitrary messages using a data channel is not needed, so the media server may also be classified as a WebRTC gateway.

Another good option would be the ability to support the session instantiation protocol (SIP), to connect to SIP compatible devices and connect them directly to WebRTC compatible devices.

Stream handling

The media server needs to save streams to a file system, both to local disks and network shares. That way the participants may download the recorded conference after it is finished.

In order to handle and save the media streams, the RAM should be generously sized and fast, and also the writing speed of the connected hard drives is crucial. This server will not be cheap if it needs to support more than three or four conversations at the same time.

Transcoding capabilities

As WebRTC defines H.264 and VP8 as mandatory codecs[22], the media server needs to support both. For H.264 a license should be obtained[2].

In the future the respective succeeding video codecs, VP9 and H.265 need to be supported, too. The transcoding of these formats, and maybe some additional codecs that are used in the communication of legacy SIP communication systems needs to be supported in real time without adding additional delay to the streams.

4.2 Signaling application

The signaling application can be divided into multiple parts:

Some are only relevant on the server side of the signaling application, some are only relevant for the client, and some parts are needed both on the server and the client side, but need different implementations, like the messaging system.

If the messaging system is to be used for the signaling of WebRTC peer connections, it needs to be implemented so that arbitrary messages can be exchanged between the clients and the signaling application[3]. It is used to supply all clients with needed SDP information and ICE candidates of the other peers or WebRTC endpoints on the media server to establish peer connections to the media server and to receive media streams.

In order to create a WebRTC connection, ICE candidates need to be passed between two peers. The fastest way would be to store the candidates in the signaling application so if another user tries to connect, he could receive the gathered candidates and try to establish a connection.

[2]Cisco has open sourced their implementation of H.264 under a permissive BSD license and pays the needed licensing fees to MPEG LA if the compiled binary is used[23]

[3]Right now text-based SDP is used, but in the future a JavaScript Object Model will be used[9].

Also the SDP offer of each participant can be saved in the signaling application, which also contains some ICE candidates.

The clients should be able to send and receive chat messages in the group. This should work without creating new connections. Each client should send the messages to the signaling application and it should relay the messages to the other participants. That way, a persistent chat log could be created instead of a system, where the user only sees messages that were posted to a room while he was inside that room.

4.2.1 Connecting the media server and a participant

The media server should act like a normal WebRTC device and use a WebRTC peer connection to connect to a participant. This includes that it must receive an SDP offer, generate an answer and send it to the other peer. This sending of offer and answer messages is done by the signaling application.

The signaling application should be enabled to create WebRTC endpoints on the media server for the participants of a conference to connect to. A wrapper for remote procedure calls is needed.

On the server side a system that supports rooms or groups conference is needed.
The state of participants needs to be kept, too. This includes a display name, a unique user id, and an optional room id.

One room id will suffice because a use case where one person might speak in two or more conferences is not considered feasible.
The room id could also be used to provide a way for the users to download the recorded conferences after they have finished.

Additionally, the WebRTC endpoints that were created for a user on the media server should be kept in order to close and release them after a user has exited from a conversation to keep unnecessary memory consumption on the media server to a minimum.

4.2.2 Client side

On the clients' side a way is needed to display chat messages and to show videos.
Also the client needs to be able to create rooms (or group conferences), and to enter or leave existing rooms.

User interaction flow

First a user enters the main page of the signaling application. A list of rooms is displayed that the user may join or additionally he may create a new one.

After the user clicked on the room, the room view is displayed. There he may choose to start the capturing of a video from his camera and microphone. Once the WebRTC peer connection to the media server is established, he should be displayed in the room for other users to see. Also his captured stream should be displayed on the page, but muted.

As other users join the room, their name should be displayed and once they also transmit their streams, they should also be displayed in the room.

The user may also hit the leave button to exit the conference and continue on to the main page to join another room.

4.2.3 Additional tasks in a production environment

In a production environment, further things should be considered, for instance could the signaling application be used for load balancing capabilities and to ensure security measures.

The signaling application is of light weight and may also be used as a load balancer to distribute participants to multiple media server instances.

The signaling application and its load balancing abilities could also be used to protect the media server from malicious (D)DOS attacks.

4.3 STUN and TURN server

For WebRTC to work reliably, a STUN server is needed to provide the public IP addresses of the participants.
To use that service, a free STUN server[4], a bought one or a self hosted might be used.
As written before, according to a study released by Bryan Ford**??**, in about 18% of the tested networks, a TURN server is needed to establish a peer to peer connection that may be used for WebRTC.
If the STUN server uses the same IP address as the media server, it is even more unlikely that a TURN server is needed, as shown in section 3.1.3 on page 15.

The best option would be to use a server that is both a TURN and a media server, but as of February 2015, no such option implementation exists yet.
To use two different applications on one machine to achieve this goal would use a lot of resources[5]and most likely not work in a reliable way.

[4]A list of free to use STUN servers[24]

[5]The amount of data that needs to be processed on the machine would be be doubled, because the TURN server forwards the encrypted data stream in a new encrypted connection and the media server needs to decode the new stream and send it back to the TURN server where it will be embedded into another connection again.

Part II

Practical part

Chapter 5

The prototype

The prototype of a WebRTC multipoint conferencing solution for more than four participants that allowed recording using a Media Server was implemented as simucos - **si**mple **mu**ltipoint **co**nferencing **s**olution.
As depicted in figure 5.1 it consists of a media server, a signaling application on the server and a web application on the client's side.

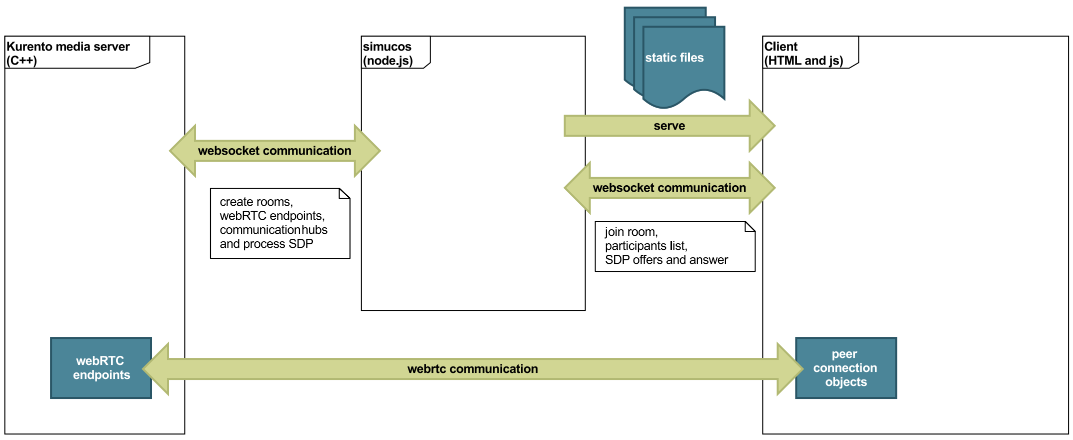


Figure 5.1: Prototype application overview

5.0.1 Copyright

The rights to simucos belong to BeamYourScreen GmbH and the source code is not enclosed in this document.
General information about different parts of the solution, lessons learned and challenges during the development are documented in this part of the thesis.

5.1 Media server

Main focus of this thesis is to find out how multipoint conferencing with recording can be achieved for multiple participants using WebRTC.

That is why the implementation of a new media server is out of the scope of this thesis and it was decided at the beginning to evaluate existing media server solutions and use one.

5.1.1 Comparison of existing media server solutions

Many different possibilities were evaluated and a short listing of that evaluation is located in the appendix on page 51.
Most of those solutions may be deployed for self-hosting, but also a number of cloud based hosted services were investigated and documented.

5.1.2 Selection of Kurento media server

After careful consideration, the **Kurento media server**[25] was chosen because it supports transcoding between the two required codecs H.264 and VP8[22], and is released under the permissive LGPL license that allows to use it unaltered in a commercial environment.

It also supports the concept of pluggable media pipelines that allows the implementation of both mixed and relayed media servers[1], and extensive documentation.

Also a JavaScript API and interface implementation is included, that allows the same functionality as connecting to the media server from a Java Application Server, without needing to implement a new wrapper for remote procedure calls.

The Kurento media server can be built on any Linux machine and a pre-compiled Debian package of the stable version is also available, which makes it very easy to set up.

In the programmed prototype, the media server and signaling application are running on the same machine. The media server is started as a system service and the signaling application is started as a Node.js application.

5.2 Signaling application

The signaling application was written for Node.js[2] and uses the following node modules and JavaScript components.

Table 5.1: Third party code used in the signaling application

Component	License	Description
Adapter.js[3]	Apache 2.0	Unifies the browser interfaces for WebRTC functions
Bower[4]	MIT	A package manager for Node.js and the web
Express[5]	MIT	Web framework for node.js - used to serve files to client
Gulp[6]	MIT	Build tool to minify code and styles for production
Kurento-client[7]	LGPL	Needed for RPCs to the Kurento media server
Q[8]	MIT	Library for promises
Socket.io[9]	MIT	WebSocket server and client

5.3 STUN server

For this prototype a list of free STUN servers[24] was used instead of setting up an own STUN server.
In production several STUN servers should be used, or one STUN server should be deployed to each instance of the media server in order to be certain about the quality of service.

[1]Mixed and relayed media servers are defined in section 3.2 starting on page 21
[2]Node.js[26] is an application runtime and web server.
[3]Adapter.js: `https://github.com/Temasys/AdapterJS`
[4]Bower: `http://bower.io/`
[5]Express: `https://github.com/strongloop/express`
[6]Gulp: `https://github.com/gulpjs/gulp`
[7]Kurento-client: `https://github.com/Kurento/kurento-client-js`
[8]Q: `https://github.com/kriskowal/q`
[9]Socket.io: `http://socket.io`

Chapter 6

Kurento media server

This chapter is a summary of the official Kurento 5.1.0 documentation[27] and contains information how the media server works and explains the concept of pluggable media pipelines that allows for a lot of flexibility in implementation.

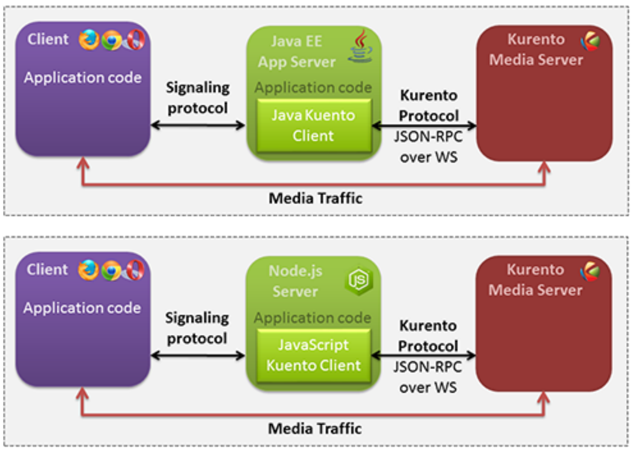


Figure 6.1: Kurento clients and Kurento media server[27, chapter 3]

The Kurento media server has two different clients that may be used in a signaling application: One for the Java EE environment, and one for Node.js. Both allow to trigger remote procedure calls on the media server.

This allows to create media pipelines that contain several components. Figure 6.2 shows a pipeline that is used for a two-way WebRTC conference that also records each participant's stream to a file.

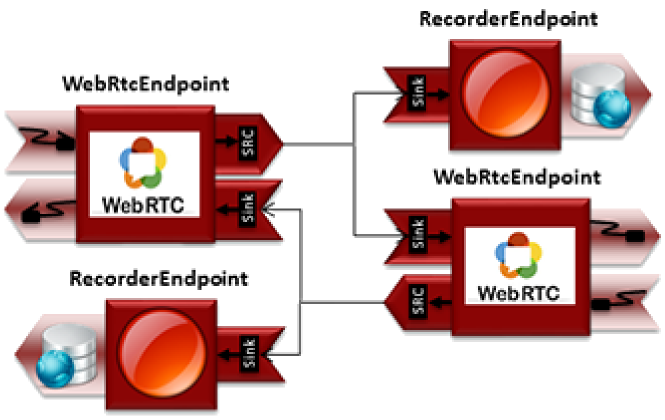


Figure 6.2: A recording pipeline for a two person WebRTC chat[27, chapter 9]

Each participant sends his stream to a WebRtcEndpoint in the media pipeline and receives another stream back. The output point of the incoming transmission on a WebRtcEndpoint is in both cases connected to a RecorderEndpoint and the input point of the outgoing transmission of the other WebRtcEndpoint.

Also other components exists, for instance the Composite hub can be used to combine multiple streams into a video wall and send this new video stream to multiple participants.
Media pipeline components

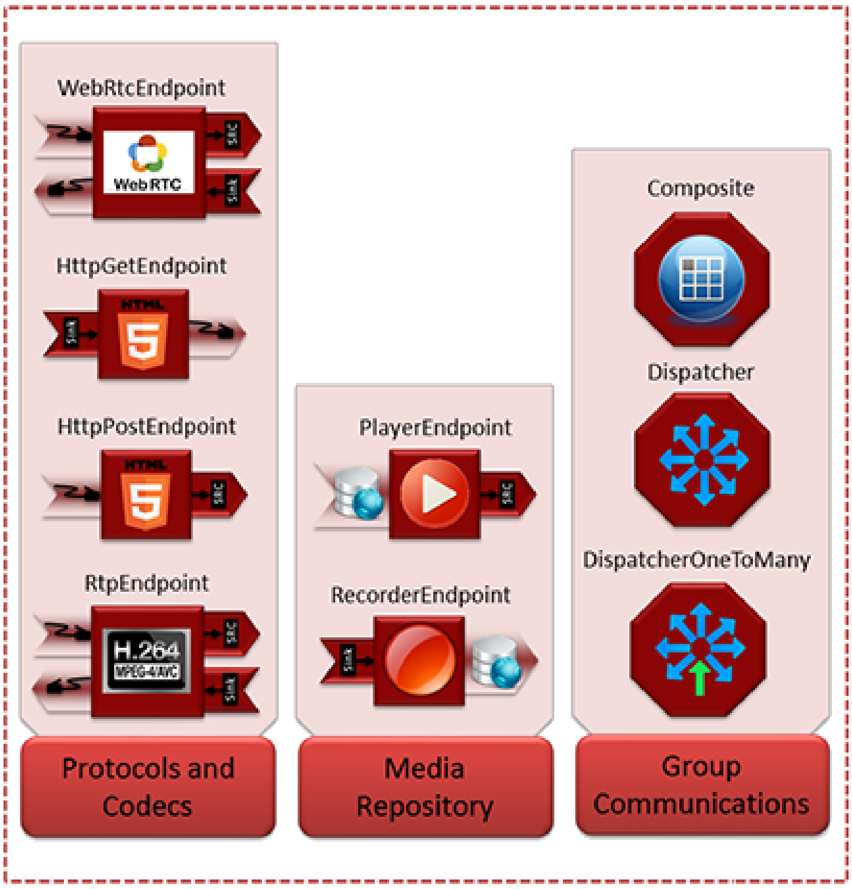


Figure 6.3: Several components usable on media pipelines[27, chapter 3]

Chapter 7

Signaling application

The structure and implementation conform to an event-driven architecture[28, chapter 2]. This allows for a high concurrency of requests with high throughput while keeping the amount of side effects as low as possible.

For tighter coupled components, like the inner workings of the main **serverState** module, promises were used instead of the Node.js standard way of using error-first callback functions. Because promises are not yet part of all ECMAScript implementations, the Q library[29] was used. It allows to use promises and also provides a way to wrap error-first callback functions into promise objects conforming to the coming standard definition of promises[1].

7.1 Evented architecture

The signaling application is split into multiple independent node modules that communicate using events[2].
The three major modules and the events they use for communication are depicted in figure 7.1[3].

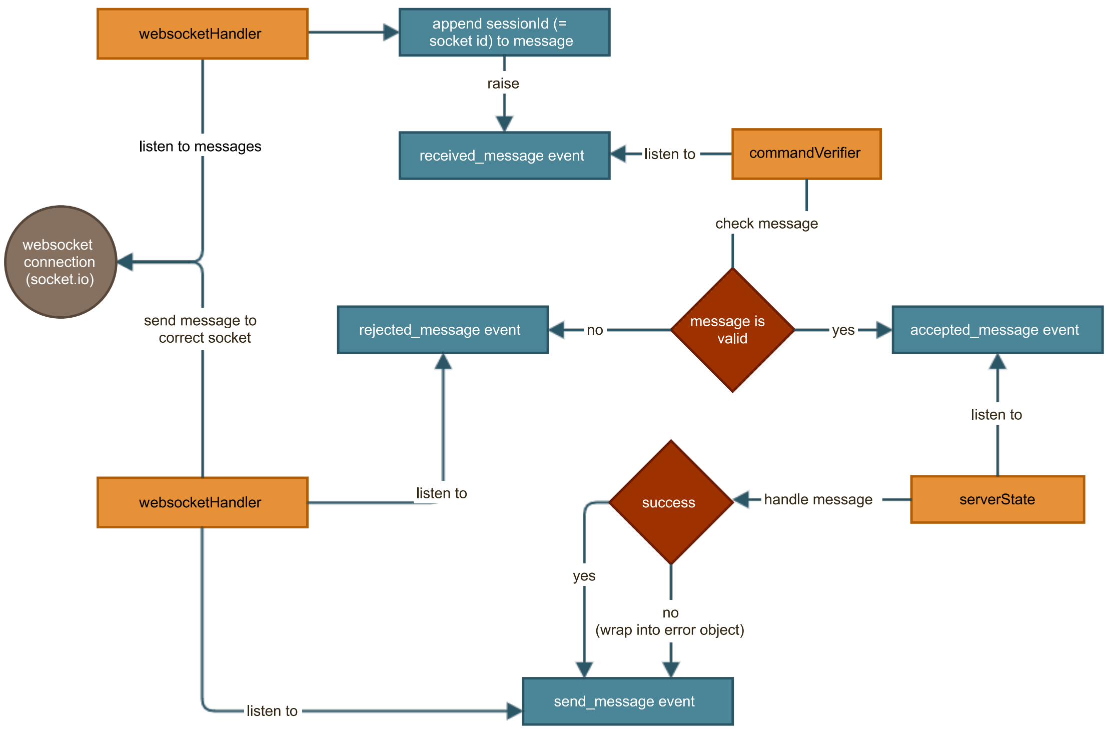


Figure 7.1: Evented architecture of the simucos signaling application

[1]Promises are part of ECMAScript 2015[30, section 18.3.18], formerly known ES Harmony.
[2]For events the Node.js EventEmitter [31, chapter 4] is used
[3]The diagram was created with the web tool gliffy `http://www.gliffy.com/`

7.1.1 websocketHandler

All communication between the signaling application and the client, and between the signaling application and the media server is done using WebSockets as defined in [RFC 6455]. The functionality is located in the **websocketHandler** module, which abstracts the socket.io[32, chapter 4] library and raises `received_message` events when one of the WebSocket connections receives data.

Each event contains the sending client's unique id to identify the source and to allow a truly asynchronous event propagation.

7.1.2 commandVerifier

The **commandVerifier** module listens to these events and verifies their structure and content. Depending on the verification of the message either a `rejected_message` or a `accepted_message` event are triggered.

Each rejected message is picked up by the **websocketHandler** and is used to send an error message to the client that tried to send that message. An accepted message is read by the main module, called **serverState**.

That way both partial messages and messages with malicious intent are not passed on to the main application. Also the accepted messages are copied so that only the desired properties are set, which also adds to the security of the application.

7.1.3 serverState

The **serverState** module consists of multiple sub modules that are not further described here but are separated into functions for a relayed or a mixed conversation and also abstraction layers for the media server and handling of the command messages.

After the message was parsed by the **serverState** module and the demanded actions were executed, a `send_message` event is raised that either contains an answer to the client or an error object that specifies a problem that occurred during executing the desired command.

The **websocketHandler** listens to these events and passes them on to the destination client.

7.2 Example: Start a conversation

Figure 7.2[4]shows an overview of the sequence of events and user interactions that are needed to start a conversation in simucos and connect a clients' WebRTC browser to a WebRTC endpoint on the Kurento media server.

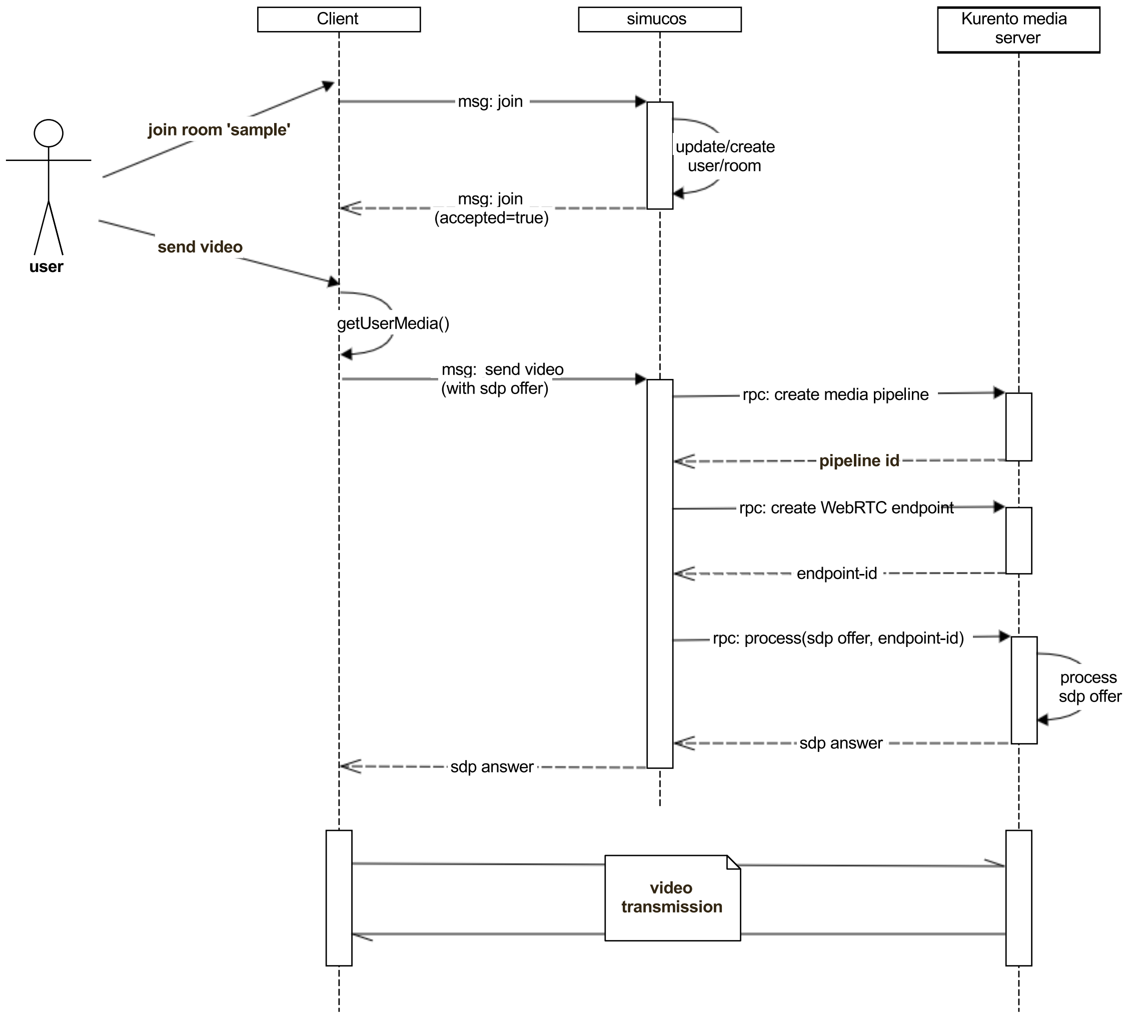


Figure 7.2: Sequence diagram: How to start a conversation in simucos

Error handling and the verification of commands is omitted for brevity.

7.2.1 Step 1: Join a room

Simucos uses the concept of conversation rooms to support multiple conferences at the same time. First the user needs to join a room by selecting one of the existing rooms that are displayed on the landing page or by creating a new one. A third option is to join or create a room using a direct link.

If either the room or the user does not yet exist in the simucos database, it is created.

After the user was successfully added to the room, it and all participants in it are displayed to the user. After that the user may choose to send a video to the room.

[4]The diagram was created with the web tool gliffy `http://www.gliffy.com/`

7.2.2 Step 2: Send a video stream

The user triggers the WebRTC getUserMedia()[5]request. After the user has chosen camera and/or microphone an SDP offer is generated. That offer is sent as part of a send video message to simucos.

The signaling application checks if a media pipeline was already saved to its database for the user's conversation room. If not, it triggers a remote procedure call on the media server to generate a media pipeline and stores its id after it was successfully created for further use.

Then a WebRTC endpoint is created on that media pipeline and its id is also stored. After that the SDP offer is sent to the endpoint where it is processed to generate an SDP answer that is returned to simucos and then delivered to the client.

7.2.3 Step 3: Communication stream

After that the client and the WebRTC endpoint on the media server negotiate a WebRTC peer connection and the client sends his video stream to the media server.

If the conference type of the room was set to mixed, the media server sends the mixed video of all participants back to the user. That way, only one connection is needed between the client and the media server.

If a relayed conference was chosen, the media server sends the user's own stream back while in development mode or it does not return a stream in production mode. To display the other participants in a relayed conversation, the client starts a new WebRTC peer connection for each other participant and connects to their WebRTC endpoints on the media server.

[5]This call returns a media stream for use in WebRTC[4, section 10.2.1]

Chapter 8

Benchmarking WebRTC bandwidth usage

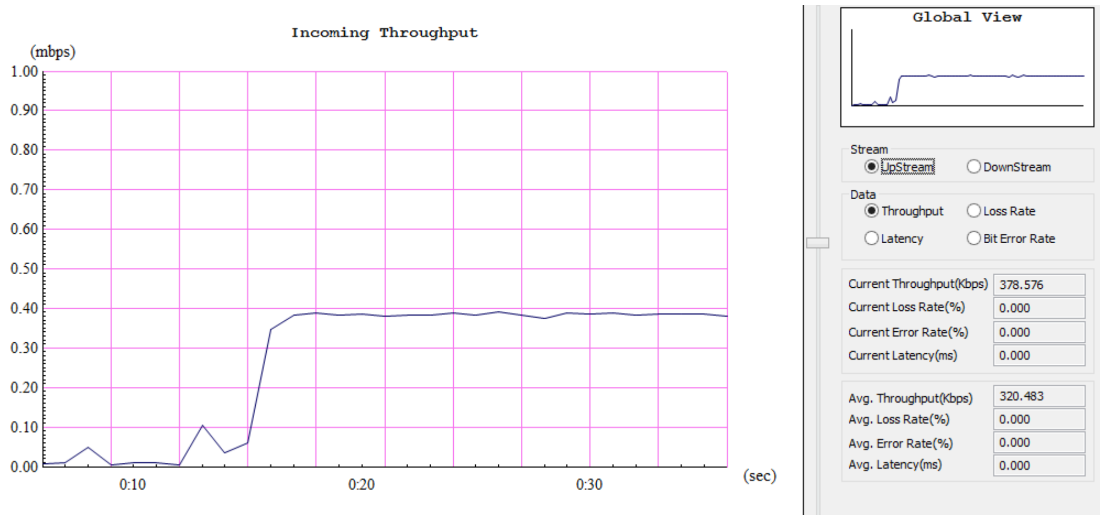
The implemented simucos prototype was used to conduct benchmarks on the behavior the WebRTC video stream transmission.

Several common video resolutions with aspect ratio of 4:3 or 16:9 ranging from 176x144 (QCIF) to 1080x720 (HD 720p) pixels were chosen and tested with different bandwidth limitations.

The gathered data is used to create a benchmark and formulate recommendations for the different architecture options.

8.0.4 Limiting the bandwidth

In order to limit the bandwidth on the testing device, the Network Emulator Toolkit (NEWT)[1]was used.



Stream: [✓ UpStream] DownStream
Data: [✓ Throughput] Loss Rate Latency Bit Error Rate

Current Throughput(Kbps)	378.576
Current Loss Rate(%)	0.000
Current Error Rate(%)	0.000
Current Latency(ms)	0.000
Avg. Throughput(Kbps)	320.483
Avg. Loss Rate(%)	0.000
Avg. Error Rate(%)	0.000
Avg. Latency(ms)	0.000

Figure 8.1: A WebRTC video stream at 176x144 with 15fps limited to 384kbps[2]

Figure 8.1 shows how the WebSocket connection is established after eight seconds (the first spike), the SDP information is exchanged after 13 seconds, and after 15 seconds the video stream is sent over the connection and effectively limited to 384 kbps.

Another feature of the WebRTC video transmission using VP8 can be seen in figure 8.2 between 16 and 26 seconds, where the codec bit rate is adapted multiple times to the available bandwidth depending on the quality of the received video. After 26 seconds, WebRTC will still try to increase the quality, but the bandwidth is limited to 512 kbps, resulting in a rippled continued graph.

[1]A program that was created my Microsoft for Visual Studio[33] that allows to limit bandwidth, simulate high packet loss or high strain on network devices in windows by hooking up to the network interface driver.

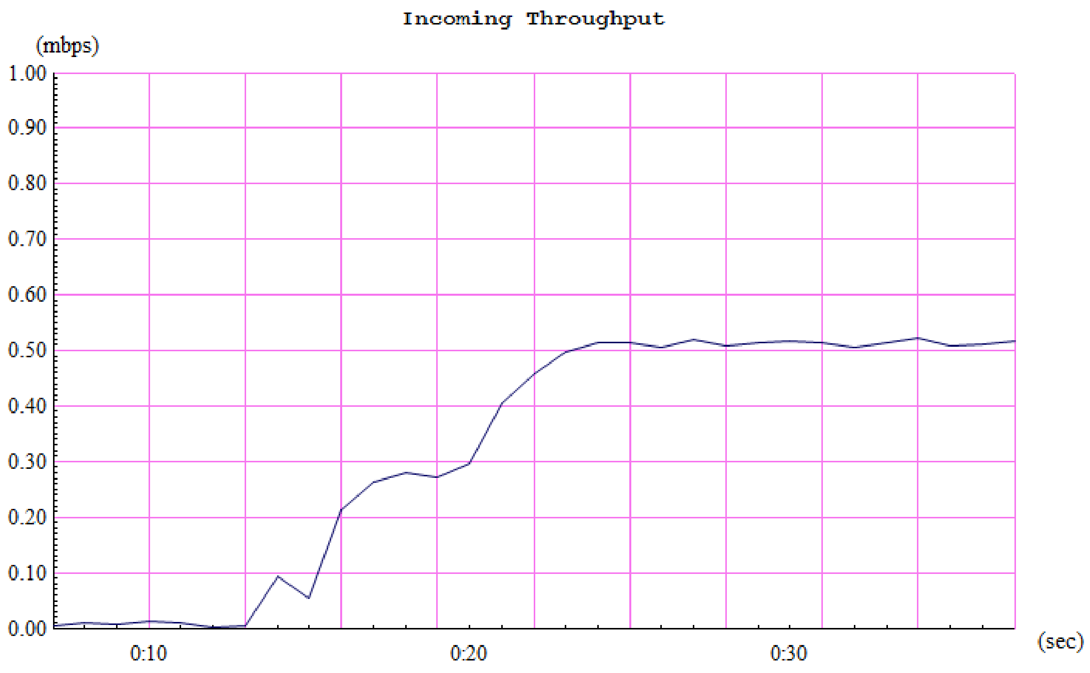


Figure 8.2: A WebRTC video stream at 352x288 with15fps limited to 512kbps[2]

8.0.5 Bandwidth recommendations for one WebRTC stream

The bandwidth recommendations depicted in table 8.1 resulted of the gathered bandwidth data, where for each resolution at a constant rate of 15 frames per seconds, and the bandwidth limitations 200 kbps, 384 kbps, 512 kbps, 1024 kbps, 2048 kbps, 4096 kbps were used.

Each conference was conducted for the duration of three minutes and the transmitted video was always similar with one person sitting in front of the camera and waving a hand slowly. The video was recorded on the media server.

After that, each recorded video was analyzed and the number of stream freezes, transmission interruptions was noted down and used to find bandwidth recommendations for the specific resolutions.

Table 8.1: Bandwidth recommendations with different video resolutions

Entry	Resolution	recommended	ok[2]	maximum[3]
QCIF	176x144	384kbit/s	200kbit/s	700kbit/s
CIF	352x288	700kbit/s	384kbit/s	2000kbit/s
VGA	640x480	1024kbit/s	512kbit/s	2100kbit/s
HD 720p	1280x720	1900kbit/s	1024kbit/s	2500kbit/s
HD 1080p	1920x1080	-	-	-

[2]Created with the Network Emulator Toolkit
[2]Minor hickups in the stream at maximum one stream freeze per minute
[3]Maximum used bandwidth in this configuration

8.0.6 Needed bandwidth for different conference sizes

These figures are based on the recommended bandwidth for each video resolution with a fixed rate of 15 frames per second and calculated for different numbers of participants with the different multipoint architectures peer-to-peer (p2p), relayed and mixed.

Table 8.2 shows the bandwidth that is needed on each client's side for a conversation, and table 8.3 shows the bandwidth that is needed on the server side for the whole conversation.

Table 8.2: Client side:
Needed bandwidth depending on conference size and multipoint architecture

	QCIF (168x144)			
participants	2	4	7	10
P2P	0.8 Mbit/s	2.3 Mbit/s	4.6 Mbit/s	6.9 MBit/s
Relay	0.8 Mbit/s	1.5 Mbit/s	2.7 Mbit/s	3.8 MBit/s
Mixer	0.8 Mbit/s	0.8 Mbit/s	0.8 Mbit/s	0.8 MBit/s

	CIF (376x288)			
participants	2	4	7	10
P2P	1.4 Mbit/s	4.2 Mbit/s	8.4 Mbit/s	12.6 MBit/s
Relay	1.4 Mbit/s	2.8 Mbit/s	4.9 Mbit/s	7.0 MBit/s
Mixer	1.4 Mbit/s	1.4 Mbit/s	1.4 Mbit/s	1.4 MBit/s

	VGA (640x480)			
participants	2	4	7	10
P2P	2.0 Mbit/s	6.1 Mbit/s	12.3 Mbit/s	18.4 MBit/s
Relay	2.0 Mbit/s	4.1 Mbit/s	7.1 Mbit/s	10.2 MBit/s
Mixer	2.0 Mbit/s	2.0 Mbit/s	2.0 Mbit/s	2.0 MBit/s

	HD 720p (1080x720)			
participants	2	4	7	10
P2P	3.8 Mbit/s	11.4 Mbit/s	22.8 Mbit/s	34.2 MBit/s
Relay	3.8 Mbit/s	7.6 Mbit/s	13.3 Mbit/s	19.0 MBit/s
Mixer	3.8 Mbit/s	3.8 Mbit/s	3.8 Mbit/s	3.8 MBit/s

8.0.7 Recommendations for different network environments

Several conclusions can be drawn from the figures calcuated in table 8.2[4]:

In a **3G** network, a conversation between two participants, or a mixed conference might work with all tested resolutions.

In a **4G** network, a peer-to-peer conversation with four participants might work with a resolution lower than VGA, a relayed conference might work with up to six participants on a VGA resolution.

A p2p conversation with seven participants using a VGA resolution will most likely not work when using a **WLAN 802.11b** router.

A **WLAN 802.11g** network is needed for a relayed conversation with 10 participants, while a p2p conference with seven participants with a HD 720p resolution will need a higher bandwidth.

Table 8.3: Server side:
Needed bandwidth depending on conference size and multipoint architecture

	QCIF (168x144)			
participants	2	4	7	10
Relay	1.5 Mbit/s	6.1 Mbit/s	18.8 Mbit/s	38.4 MBit/s
Mixer	1.5 Mbit/s	3.1 Mbit/s	5.4 Mbit/s	7.6 MBit/s

	CIF (376x288)			
participants	2	4	7	10
Relay	2.8 Mbit/s	11.2 Mbit/s	34.3 Mbit/s	70.0 MBit/s
Mixer	2.8 Mbit/s	5.6 Mbit/s	9.8 Mbit/s	14.0 MBit/s

	VGA (640x480)			
participants	2	4	7	10
Relay	2.0 Mbit/s	16.3 Mbit/s	50.2 Mbit/s	102.4 MBit/s
Mixer	2.0 Mbit/s	8.2 Mbit/s	14.3 Mbit/s	20.5 MBit/s

	HD 720p (1080x720)			
participants	2	4	7	10
Relay	7.6 Mbit/s	30.4 Mbit/s	93.1 Mbit/s	190 MBit/s
Mixer	7.6 Mbit/s	15.2 Mbit/s	26.6 Mbit/s	38 MBit/s

[4]Information about data rates in Wifi[11, chapter5-7], 3G[34] and 4G[35] networks.

Chapter 9

Lessons learned

9.1 Promises

Promises are a part of the coming ECMAScript 2015 standard[30] to allow developers to implement data flow differently from using callback functions for dependency injection.

9.1.1 Example with promises

Listing 9.1 shows code that is used similarily in simucos to connect a user's WebRTC peer connection to a WebRTC endpoint on the media server and uses the kurento-client[36] JavaScript module.

Several functions need to be called after another while executing them asynchronously. First, the user's WebRTC endpoint needs to be connected to the media server, then the recording of the video stream is started and in the end the user receives a message containing the SDP answer to actually start sending data to the media server.

```
1  function connectUserToRelayedConference(user, sdpOffer) {
2    connectToMediaServer(user, sdpOffer)
3    .then(recordConnection(user.webRtcEndpoint))
4    .then(sendMessageTo(user, sdpAnswer))
5    .fail(function (reason) {
6      console.log("Error handling");
7    });
8  }
```

Listing 9.1: Example for promises

On line 2 in listing 9.2 the Q library[29] is used to create a new promise object. On line 7 an asynchronous call using an error-first callback function to retreive or create a media pipeline on the media server is started. Before this function is called on line 8, the promise object is returned on line 33.

When the callback function is called, a test is used to check for errors during that function. If no errors occur, the promise object is resolved on line 24 and returns the SDP answer.

If an error occurred a new exception is thrown in the context of the `connectToMediaServer` function that is caught in line 33 and rejects the promise object.

If the promise object was resolved, the function proceeds to line 4 of listing 9.1 and continues the asynchronous function calls.

If the promise object was rejected, or any other promises like `recordConnection` or `sendMessageTo` are rejected, the `fail` method is invoked in line 6 of listing 9.1 and centralized error handling can be used there.

```
1  function connectToMediaServer(sendingUser, sdpOffer) {
2    var deferred = Q.defer();
3    var roomId = sendingUser.roomId;
4    var sourceEndpoint = sendingUser.webRtcEndpoint;
5
6    try {
  getPipeline(roomId, function (error, pipeline) {
    if (error) { throw error; }

    pipeline.create('WebRtcEndpoint', function (error, endpoint) {
      if (error) { throw error; }

      endpoint.processOffer(sdpOffer, function (error, sdpAnswer) {
        if (error) { throw error; }

        sourceEndpoint.connect(endpoint, function (error) {
          if (error) { throw error; }

          var response = {
            success: true, message: 'Everything is fine',
            sdpAnswer: sdpAnswer
          }

          deferred.resolve(response);
      });
    });
  });

  } catch (err) {
    deferred.reject(err);
  }

  return deferred.promise;
});
```

Listing 9.2: Example to return a promise object from callbacks

9.1.2 Example with callback functions

The same example function can also be written using callback functions and would look like listing 9.3 to inject the functions into the callbacks.

```
function connectUserToRelayedConference(user, sdpOffer) {

  return connectToMediaServer(user, sdpOffer, funtion (error, sdpAnswer) {
    if (error) { console.log("Error handling"); return; }

    return recordConnection(user.webRtcEndpoint, function (error) {
      if (!error) {
        sendMessageTo(user, sdpAnswer);
      }
      return;
    })
  });
}
```

Listing 9.3: Example for error-first callbacks

The function `connectToMediaServer` from listing 9.2 would not change much as seen in listing 9.4, but it would not return a promise object and isntead call a callback function that is passed into it.

```
function connectToMediaServer(sendingUser, sdpOffer, callback) {
  var roomId = sendingUser.roomId;
  var sourceEndpoint = sendingUser.webRtcEndpoint;

  try {
  getPipeline(roomId, function (error, pipeline) {
    if (error) { return callback(error); }

    pipeline.create('WebRtcEndpoint', function (error, endpoint) {
      if (error) { return callback(error); }

      endpoint.processOffer(sdpOffer, function (error, sdpAnswer) {
        if (error) { return callback(error); }

        sourceEndpoint.connect(endpoint, function (error) {
          if (error) { return callback(error); }

          var response = {
            success: true, message: 'Everything is fine',
            sdpAnswer: sdpAnswer
          }

          return callback(null, error);
      });
    });
  });

  } catch (err) {
    return callback(err);
  }
});
```

Listing 9.4: Example for error-first callbacks

9.1.3 Benefits from using promises

These simple examples show the beauty of the cleaner syntax that is possible using a promise based API when compared to one based on callback functions.

WebRTC will also benefit from this and will allow even faster development cycles while simplifying the code structure and improving its readability.

9.1.4 Combining the promise and error-first-callback approach

The Q library[29] also allows to create functions or modules that can both return a promise object and call a callback function.
This approach was actually used in simucos to allow other modules to keep using their approach when using the simucos module.

A simple example for this functionality is provided in listing 9.5, where the Node.js File API is used to read a configuration file from disk.

```
//dual-module.js
var Q = require('q'), fs = require('fs');

module.exports = {
  readConfiguration: function (file, callback) {
    var deferred = Q.defer();

    fs.readFile(function (error, data) {
      if (error) deferred.reject(error);
      else deferred.resolve(data);
    });

    deferred.promise.nodeify(callback);
    return deferred.promise;
  }
}
```

Listing 9.5: Dual module that supports promises and callbacks

The function may then be used either with a promise-based approach as in listing 9.6 or using a callback-based approach as in listing 9.7.

```
var DualModule = require('dual-module');

DualModule.readConfiguration('conf.json')
.then(function (data) {
  //do something with the configuration data
})
.fail(function (reason) {
  //initialize default values
})
```

Listing 9.6: Using a dual module with promises

```
var DualModule = require('dual-module');

DualModule.readConfiguration('conf.json', function (error, data) {
  if (error) {
    //initialize default values
  }
  else {
    //do something with the configuration data
  }
});
```

Listing 9.7: Using a dual module with callbacks

Chapter 10

Conclusion

10.1 Using WebRTC

It is shown, that WebRTC is a very mature technology despite its young age and the first approved W3C Working Draft was released in February 2015.

The technology is needed, and it definitely has disruptive qualities as it enables web developers to create communication applications in a very short time. Using preexisting VoIP technology, expensive hardware and a lot more experience and time would have been needed to create a similar solution.

The future will be very interesting concerning this technology, and especially the additional features that are discussed for ORTC[9]: Allowing to exchange the unwieldy SDP messages with concise JSON objects.
Also using stream quality data to further adapt the bit rate for the video codec and increase the options for developers to access lower level functions in order to detect speaking participants or to suppress background noise will even add to WebRTCs importance.
If WebRTC is adopted by VoIP engineers they will even be able to keep a substantial part of their current market and create better solutions at the same time.

Another good feature that was already introduced into the current Working Draft is the switch from inversion-of-control callbacks for the browser API to a promised-based approach, which is not yet implemented into Mozilla Firefox nor Google Chrome.

10.2 Working with the Kurento media server

The Kurento media server is a very sophisticated piece of software that is in active development and provides extensive documentation and an active community.
The API is very well thought out and even if the Java EE implementation is still needed when trying to integrate SIP, the JavaScript Client[36] allows for rapid development and prototyping without many obstacles.

10.3 Goals

Most of the goals for this thesis were achieved with the simucos prototype:
Conferences can be recorded on the media server as VP8 or h.264 video files.
Many clients may participate in a conference, the highest number tested was 25 in a mixed conference, and 14 in a relayed conference.
Participants in restrictive network environments can connect to a conference using a STUN or TURN server.
Only the optional goal to add support for SIP clients into simucos was not implemented due to the chosen Node.js architecture.

10.4 Future development

Right now simucos is being integrated into another prototype application that uses WebRTC peer-to-peer conferences. The new prototype will be able to start conferences either using a peer-to-peer full mesh network or a media server and also to switch the conference type.

10.4.1 Switching conference to the media server

The simplest option would be to switch the conference to a media server once a participant presses a record button in the conversation.

10.4.2 Switching conferences as needed

An interesting addition to the prototype would be to start a conversation as a full mesh. As more participants join into the conference, the conference switches first to a relayed and then to a mixed conference on the media server.

This switch could occur either based on the number of participants alone or on the conference size and the stream quality each participant achieves.

For the latter, the statistics module of WebRTC could be used and the signaling application would calculate the overall quality of the conference. Also this mechanism could be used to reduce the video resolution or bit rate, but functions like this are not yet part of the WebRTC specification[10]. These are considered for inclusion into ORTC[9].

Also the conference should switch back to a full mesh architecture, if the number of participants has decreased below the threshold or only participants with sufficient bandwidth and processing power remain in the conference.

Part III

Appendix

Self hosted solutions - Comparison table

This list was composed in October and November 2014, and some informations in it might be already outdated.

Media Servers		**Doubango Telepresence**	**FlashPhoner**	**Jitsi**	**Kurento Media Server**	**Licode**	**Medooze and Mobicents**	**NG Media Server**	**Powermedia XMS**
		doubango		Libjitsi, Video Bridge, JitMeet		Lynckia, Erizo	Medooze MTU and Mobicents AppServer (SIP)		Dialogic
short	**blocker**	no	audio only recording	no	no	no	no	no	no
	pros	- transcode between VP8 and H.264 - Commercial license available	Works together with Wowza (already in use)	- OpenFire plugin (magnocall interoperability) - 10 persons chatting simultaneously on 1.6Ghz 1 Core	- Kurento Application Server is supplied - uses JSR309 (Media Server Control API) - Pluggable pipelines (filter, post-process and transcode media streams) - connect with SIP using Kurento Application Server - Receive-only streaming with http	- MIT license, changes are allowed and don't need to be published - Allows to record each participant stream - Allows to send messages over data stream - Erizo is split up in 3 parts and may be run on different machines (Controller, Agent, JS) - play recorded files into conversation	- JSR309 (Media Server API) - Commercial License available	-	- JSR309 (Media Se API) - REST Interface - Audio features: Voi activity detection, sil suppression, comfor noise generation
	cons	- No information about benchmarks - GPL - codecs need to be built on/for destination machine (licensing?) - transcoding uses ffmpeg (Doubango discourages using ffmpeg due to licensing issues) - one encoder per codec, transcoding for low bandwidth is not supported (one setup for all encoders)	- No information on benchmarks - expensive	- LGPL - Application Server needed - only works on Chrome	- LGPL - only Media Server, Application Server is needed - No direct connection with SIP	- No mixing functionality - No transcoding of streams for low bandwidth - No SCTP - No direct connection with SIP clients - Application Server is needed	- Needs Application Server - GPL - MCU Media Server is only available in GPL	-most information is either vague or in french (most are both)	- Needs Application Server
Goals									
record		yes to local file, using ffmpeg	audio only	yes	yes VP8 and H.264	Yes, each participating stream is recorded separately on the server (as mkv with VP8 and Opus). Recording is started by the participant.	yes	Yes, but no further information	yes
8+ simult. participants			no information	24+ possible		yes			yes, see benchmark
ICE (stun, turn, negotiation, none)		negotiation	STUN, TURN, negotiation	negotiation	negotiation	negotiation	STUN, negotiation	no?	negotiation
Signaling		yes (TLS and Websockets with TLS)	SIP 2.0 rfc3261 RTP rfc3550 SDP rfc4566	yes (Websockets)	no (v5), yes (v4)	Yes, Erizo API also publishes Signaling API	yes	SIP	SIP (RFC3261) WebRTC JavaScrip RTSP

Rooms, User auth		yes, with passwords	no	yes	no (needs Application Server)	yes	no (needs Application Server)	no information	yes (API)
Connect with SIP Clients		yes	yes, with SIP/RTP	yes	no (needs Application Server)	no	no (needs Application Server)	yes	yes
Connect directly to PSTN		no	no	no	no (needs Application Server)	no	no (needs Application Server)	no	no
Costs and License		GPLv3 Or commercial license available	$56,675 (256 simult. calls) online Configurator One license for multiple Servers	free LGPL 2.1	free LGPL 2.1	MIT license	Mobicents: free (GPL) Medooze MTU: free (GPL) and commercial	Fully featured evaluation version license based on channel and feature set	Commercial license
Other Information									
Benchmark	**framework**	C++	Java and C++	Java and C++	Media Server: C++ Application Server: JEE 7	C++	none		Video: up to 450 unidirectional stream (one conference = 1 stream (mixed))
	hardware		QuadCore 2.5Ghz 4-16 GB RAM JDK 1.7 Dedicated Server (but VPS also supported)	1 Core 1.6Ghz, 4GB Ram, 4 upstreams full size, rest is mixed in (10 persons)	8GB RAM 16GB HDD Media Server: Ubuntu Linux 13.10 or newer (needs gstreamer 1.2.0) Application Server: OpenJDK 7	only works on unix based systems	none	Recommended hardware: OS: Windows Virtualization: VMWare ESXi 5.0 or higher, Microsoft HyperV Processor: QuadCore at 3GHz or higher Memory: 4 GB Disk: 500 Gb or higher Network: Ethernet 1 Gb	Minimum: < 500 cha (~100 video) Intel Xeon E5420 Quad-Core (2.50 GH 1333 MHz FSB, 80W 8 GB RAM 250 GB HDD Recommended: up t 2000 channels (~450 video) Intel Xeon X5650 D Hex-Core (2.66 GHz 1333 MHz FSB) > 16 GB RAM > 2 TB HDD
Special		- this software needs to be built on destination machine - codecs need to be installed during build (except for G.722 and G.711) - One encoder per codec (no extra encoder for Cell phone) - Usable Web-SIP-Client: http://conf-call.org/ - Presentation sharing (Screenshots of Powerpoint/LibreOffice are streamed in) - Video: Full HD (1080p) and Ultra HD (2160p), up to 120fps	No information on how the stream is recorded and in the configurator it is greyed out, but only audio is listed in the developer documentation	- Selective Forwarding Unit (many user supported, but only talking users are mixed into stream) - Only works on Chrome (multiple Tracks in one connection) - Audio is always mixed	- Works with Application Server (e.g. Kurento Application Server), which needs to implement JSR309 and Signaling - Communicates with Application Server using Kurento API (Java or JS) or Kurento Protocol (Websockets and JSON-RPC)	- Stream other resources into conversation: H.264 (RTSP), VP8+Opus (from file) - Architecture - Supports Erizo API (nodeJS) and Nuve API (python, ruby, js)	2 different Applications communicating over SIP on Websockets JSR309		Each service may be started/restarted/shu e.g. webrtc signalin msml video codecs allow parameters
			Tutorial exists on using flashphoner with Wowza	Uses XMPP for signaling and is available as an OpenFire plugin (which magnocall also uses)	Pluggable Pipelines filter, post-process and transcode media streams play recorded files	Programmed by University of Madrid	Needs a SIP application server to run, using Mobicents SIP Application Server is recommended Bitrate adaption algorithm		REST; JSR 309; J2E Converged Applicati Server (SIP Servlets MSML, NetAnn (VoiI LTE)
Codecs	**Audio**	G.711, G.722, Opus, G.711, AMR, Speex	G.711, G.729, Speex, Opus	Opus, SILK, G.722, Speex, ilbc, G.711 (PCMU, PCMA), G.729	OPUS, AMR, SPEEX	Opus	G.722, Speex, Opus	G711 (PCMA/PCMU),G729, AAC DTMF: in band (RTP-NTE rfc2833) and out of band (SIP-NOTIFY/INFO	G.711u/a, G.723, G. G.729a, G.729b, GSM-FR, GSM-EFR AMR-NB, iLBC Opus, G.722
	Video	H.264, VP8	H.263, H.263+, H.264, VP8	H.263, H.264, VP8	VP8, H.264	VP8	VP8, H.264, H.263	H.264, VP8	H.264, VP8, H.263, H.263+, H.263++ Ba Profile

Cloud based solutions - Comparison table

Contains solutions that only work on vendor-owned hardware

Media Servers		Bistri	OpenClove (ex OCX)	OpenTok	Requestec (Zenon platform)	SightCall (ex weebo)	Twilio
			OpenClove Video Exchange	TokBox	Saypage Developer API		
short	blocker	no recording, only 4 participants	no	no	competing solution	no	no video
	pros	Plugin for IE and Safari	- Flash fallback for IE and Safari - Multiplexes all streams to one for phones - view-only streams (RTSP)	- intelligent quality control (traffic shaping) - relay session (direct connection between clients may be used to reduce costs)	- Big meeting rooms - Whitboard, Screen sharing - Optimization for mobile - Web interface	- Calls from and to non-users supported - Screen sharing	- Well documente REST API - WebRTC and Flash client APIs available
	cons		- not licensable right now	- recording is only supported for routed sessions (needs to be decided when starting a session)	- competitor	- Pay per user (yearly or monthly)	- no video
Goals							
record		no	yes, download link is displayed after ending session	Recommended maximum: 5 streams; Up to 9 streams are supported (but quality might degrade). Archived streams may automatically be uploaded to e.g. amazon S3 Maximum length: 90 minutes	Yes (no info, though)	yes to cloud storage (buy separately)	10k minut free, then $0.0005 p min / mon Recorded is availabl via http download authentica needed)
8+ simult. participants		maximum of 4	9, upgradeable to 16	yes, save stream: max 9	Meeting rooms with up to 100 persons		limit: 40
ICE (stun, turn, negotiation, none)		STUN, TURN, negotiation	STUN, TURN, negotiation	STUN, TURN, negotiation	STUN, negotiation	negotiation	STUN, negotiatio
Signaling API		Proprietary (uses WebSocket)	Proprietary	Session ID and user token must be generated by external client	yes	yes	yes, base on xml
API to create rooms		yes	yes		yes	yes	yes, base on xml
API for User auth		no	yes	yes	yes	yes	no
Connect with SIP Clients		no	no		yes	no	yes
Connect directly to PSTN		no	yes (dial in costs may occur)	no	yes	no	yes

Costs and License		$30/month: 1500 calls/month $0.058 per additional call http://developers.bistri.com/webrtc-sdk/#sdk-pricin g	To be announced http://developer.openclove.com/learn	pay per minute (minute is counted for every minute over turn-server), signaling or stun is for free. E.g. If 3 persons talk on their server for 5 minutes, 15 minutes will be billed. $50 per month (includes 10k minutes - next 90k minutes: 0.475Cent / minute - prices drop to $0.004/m) Additionally: Saving media costs ~200$ per simultaneous connection (flatrate), or $0.035 per recorded minute.	???	$2 per user per month, additionally $0.10 per call from non-users Recording: $0.10 per call (on own storage)	$0.0025 p minute
Other Information							
Benchmark			9 participants 16 participants max	for archive: best quality 5 streams (max 9 - might degrade quality)	Transcoding, etc. : C/C++		
Special		Plugin for IE and Safari	- Flash fallback for IE and Safari - Multiplexes all streams to one for phones	- intelligent quality control (traffic shaping) - https://tokbox.com/#i qc (fallback: audio only and last image) - relayed sessions are cheaper than routed sessions	Mixes streams for Users with low bandwidth	- Weemo driver is downloadable if WebRTC is not supported in Browser (IE 9+, Safari 6+)	- Dynamic phone number generatio - Transcript ($0.05 per minute) (Speech t text) - WebRTC and Flash client APIs available
Codecs	Audio	Browser	Opus, G711	Opus	Browser	Browser	Browser
	Video	Browser	VP8, H.264	VP8	Browser	VP8	Browser

List of Figures

List of Tables

Listings

Glossary

1080p	A video format with a resolution of 1920x1080 pixels. Commonly referred to as Full-HD.
720p	A video format with resolution of 1280x720 pixels.
API	Application Programming Interface
DOS	Denial of Service Often also Distributed Denial of Service (DDOS) an attack that overwhelms a server by sending more requests than it can handle.
DTLS	Datagram Transport Layer Security A cryptographic protocol that is used to encrypt UDP connections.
ECMA	Ecma International, an organization dedicated to the standardization of information and communication systems.
ECMAScript	Abbreviation for a scripting language specified in ECMA-262, implemented for instance as JavaScript or JScript.
H.264	Also known as MPEG-4 Advanced Video Coding (MPEG-4 AVC) is a video format that is mandatory for WebRTC and licensed by the MPEG LA
H.264 SVC	SVC features for the H.264 video codec.
H.265	See HEVC
HEVC	High Efficiency Video Coding A successor to the H.264 video format and sometimes referred to as H.265.
IETF	Internet Engineering Task Force An open standards organization that develops and promotes Internet standards.
ITU	International Telecommunication Union A United Nations agency dedicated to information and communication technologies. Standardized for instance the G.711 audio codec[18] used in WebRTC.
Java EE	Java Platform, Enterprise Edition A java based computing platform used for Web services.
JSEP	JavaScript Session Establishment Protocol
JSON	JavaScript Object Notation Data structure format that wraps ECMAScript objects to string and back.
MCU	Multipoint Control Unit Is used in multipoint conferencing solutions. It forwards all streams to all participants.
MP4	MPEG 4 a video compression format
MPEG LA	MPEG Licensing Administration A company that administrates patent pools for multiple multimedia standards, for instance H.264 or HEVC
Node.js	An application framework often used for web applications, uses V8 and an Event Loop at its core.

ORTC	Object RTC Compatible to WebRTC, sometimes called WebRTC 1.1. Main difference to WebRTC 1.0 is to exchange text-based SDP with JSON-objects, also the web programmers have more control over the media stream.
PSTN	Public Switched Telephone Network.
RPC	Remote Procedure Call
SDP	Session Description Protocol A text based format to describe streaming media initialization parameters, both used with SIP and WebRTC.
SFU	Selective Forwarding Unit Is used in multipoint conferencing solutions. It forwards only selected streams to the participants.
SIP	Session Initialization Protocol A signaling protocol for multimedia communication, is often used for VoIP.
SVC	Scalable Video Coding, features a base layer that provides a specific resolution of a video and different sub-layers that enhance the base layer to increase the video quality, resolution or frame rate. Adapts very well to different bandwidths.
TLS	Transport Layer Security A cryptographic protocol that is used to encrypt TCP connections.
UHD	Ultra High Definition Video Also known as 4K, is used for a video format with four times as many pixels as Full-HD 1080p
V8	ECMAScript runtime developed by Google for the Chrome browser. Is also used for Node.js.
VoIP	Voice over IP Internet telephony.
VP8	One of the mandatory codecs of WebRTC, is released by Google free of licensing fees.
VP9	Successor of VP8, will feature Scalable Video Coding.
WebRTC	Web Real-Time Communication Is defined in version 1.0 and currently has the status of a W3C Working Draft
WebSocket	A protocol that provides full-duplex communication using a single TCP connection.
W3C	World Wide Web Consortium Organization that formulates standards to be used in the world wide web.
W3C WD	W3C Working Draft Multiple stages are needed to formulate a new W3C standard: First a Working Draft is formulated. After that a **Candidate Recommendation** then a **Proposed Recommendation** and in the end the document receives the status of an official **W3C Recommendation**.
WebRTC	Web Real-Time Communication An effort by the W3C, IETF and several companies to enable real time communication between browsers without needing any additional plugins. *Currently still under development.*

Bibliography

[1] R. Manson, *Getting started with WebRTC: Explore WebRTC for real-time peer-to-peer communication*, ser. Community experience distilled. Birmingham, UK: Packt Pub., 2013, ISBN: 1782166319.

[2] S. Loreto and S. P. Romano, *Real-Time Communication with WebRTC: Peer-to-Peer in the Browser.* Sebastopol, CA: O'Reilly Media, 2014, ISBN: 9781449371852. [Online]. Available: `9781449371852`.

[ECMAScript] Ecma International, *Standard ecma-262 - ecmascript language specification*, Jun. 2011. [Online]. Available: `http://www.ecma-international.org/publications/standards/Ecma-262.htm` (visited on Feb. 20, 2015).

[3] J. Uberti, C. Jennings, and E. Rescorla, "Javascript session establishment protocol", IETF Secretariat, Internet-Draft draft-ietf-rtcweb-jsep-08, 2014. [Online]. Available: `http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-jsep-08.txt` (visited on Feb. 21, 2014).

[RFC 4566] M. Handley, V. Jacobson, and C. Perkins, *SDP: Session Description Protocol*, RFC 4566 (Proposed Standard), Internet Engineering Task Force, Jul. 2006. [Online]. Available: `http://www.ietf.org/rfc/rfc4566.txt`.

[4] D. C. Burnett, A. Bergkvist, C. Jennings, and A. Narayanan, "Media capture and streams", W3C, W3C Working Draft, Feb. 2015. [Online]. Available: `http://www.w3.org/TR/2015/WD-mediacapture-streams-20150212/` (visited on Feb. 20, 2015).

[5] R. Jesup, S. Loreto, and M. Tuexen, "Webrtc data channels", IETF Secretariat, Internet-Draft draft-ietf-rtcweb-data-channel-13, 2015. [Online]. Available: `http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-data-channel-13.txt` (visited on Feb. 21, 2014).

[6] H. Alvestrand, "Overview: real time protocols for browser-based applications", IETF Secretariat, Internet-Draft draft-ietf-rtcweb-overview-13, Nov. 2014. [Online]. Available: `http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-overview-13.txt` (visited on Feb. 20, 2015).

[7] IDC, *Smartphone os market share*, 2014. [Online]. Available: `http://www.idc.com/prodserv/smartphone-os-market-share.jsp` (visited on Feb. 20, 2015).

[8] W3C ORTC Community Group, *ORTC (Object RTC) | Object API for RTC - Mobile, Server, Web*, Oct. 2014. [Online]. Available: `http://ortc.org/` (visited on Feb. 21, 2015).

[9] R. Raymond, B. Aboaba, and J. Uberti, "Object RTC (ORTC) API for WebRTC", W3C ORTC Community Group, W3C Community Draft, Jan. 2015. [Online]. Available: `http://ortc.org/wp-content/uploads/2015/01/ortc.html` (visited on Feb. 21, 2015).

[10] D. C. Burnett, A. Bergkvist, C. Jennings, and A. Narayanan, "Webrtc 1.0: real-time communication between browsers", W3C, W3C Working Draft, Feb. 2015. [Online]. Available: `http://www.w3.org/TR/2015/WD-webrtc-20150210/` (visited on Feb. 21, 2015).

[11] I. Grigorik, *High-performance browser networking.* 2013, ISBN: 1449344763.

[12] Matroska (non-profit org, *What is matroska?*, 2013. [Online]. Available: `http://www.matroska.org/technical/whatis/index.html` (visited on Feb. 21, 2015).

[13] J. Doyle and J. D. Carroll, *Routing TCP/IP*, ser. CCIE professional development. Indianapolis, Ind.: Cisco Press, 2001, ISBN: 1-57870-089-2.

[RFC 5245] J. Rosenberg, *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols*, RFC 5245 (Proposed Standard), Updated by RFC 6336, Internet Engineering Task Force, Apr. 2010. [Online]. Available: `http://www.ietf.org/rfc/rfc5245.txt`.

[14] B. Ford, *Peer-to-peer communication across network address translators*, Jun. 2005. [Online]. Available: `http://brynosaurus.com/pub/net/p2pnat/` (visited on Feb. 14, 2015).

[RFC 3489] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy, *STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)*, RFC 3489 (Proposed Standard), Obsoleted by RFC 5389, Internet Engineering Task Force, Mar. 2003. [Online]. Available: `http://www.ietf.org/rfc/rfc3489.txt`.

[RFC 5766] R. Mahy, P. Matthews, and J. Rosenberg, *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*, RFC 5766 (Proposed Standard), Internet Engineering Task Force, Apr. 2010. [Online]. Available: `http://www.ietf.org/rfc/rfc5766.txt`.

[RFC 3261] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, *SIP: Session Initiation Protocol*, RFC 3261 (Proposed Standard), Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141, 6665, 6878, Internet Engineering Task Force, Jun. 2002. [Online]. Available: `http://www.ietf.org/rfc/rfc3261.txt`.

[15] A. B. Johnston, *SIP: Understanding the Session Initiation Protocol, Third Edition*, 3rd ed. Norwood: Artech House, 2009, ISBN: 1607839962.

[16] J.-M. Valin and C. Bran, “Webrtc audio codec and processing requirements”, IETF Secretariat, Internet-Draft draft-ietf-rtcweb-audio-07, 2014. [Online]. Available: `http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-audio-07.txt` (visited on Feb. 20, 2015).

[RFC 6716] J. Valin, K. Vos, and T. Terriberry, *Definition of the Opus Audio Codec*, RFC 6716 (Proposed Standard), Internet Engineering Task Force, Sep. 2012. [Online]. Available: `http://www.ietf.org/rfc/rfc6716.txt`.

[RFC 3551] H. Schulzrinne and S. Casner, *RTP Profile for Audio and Video Conferences with Minimal Control*, RFC 3551 (INTERNET STANDARD), Updated by RFCs 5761, 7007, Internet Engineering Task Force, Jul. 2003. [Online]. Available: `http://www.ietf.org/rfc/rfc3551.txt`.

[17] O. Hersent, J.-P. Petit, and D. Gurle, *Beyond VoIP protocols: Understanding voice technology and networking techniques for IP telephony.* Hoboken, NJ: John Wiley, 2005, ISBN: 978-0-470-02362-4.

[18] “Transmission systems and media, digital systems and networks”, International Telecommunication Union, Tech. Rep. [Online]. Available: `http://www.itu.int/rec/T-REC-g` (visited on Feb. 20, 2015).

[19] &yet, *Is WebRTC ready yet.* [Online]. Available: http://iswebrtcreadyyet.com/ (visited on Feb. 20, 2015).

[RFC 5117] M. Westerlund and S. Wenger, *RTP Topologies*, RFC 5117 (Informational), Internet Engineering Task Force, Jan. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5117.txt.

[20] M. Westerlund and S. Wenger, "Rtp topologies", IETF Secretariat, Internet-Draft draft-ietf-avtcore-rtp-topologies-update-05, 2014, http://www.ietf.org/internet-drafts/draft-ietf-avtcore-rtp-topologies-update-05.txt. (visited on Feb. 20, 2015).

[21] HP, *HP Servers running HP-UX 11i*, 2013. [Online]. Available: http://h18000.www1.hp.com/products/servers/byos/hpuxservers.html (visited on Feb. 22, 2015).

[22] A. Roach, "Webrtc video processing and codec requirements", IETF Secretariat, Internet-Draft draft-ietf-rtcweb-video-04, 2015. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-video-04.txt (visited on Feb. 20, 2015).

[23] Cisco, *Openh264 frequently asked questions.* [Online]. Available: http://www.openh264.org/faq.html (visited on Feb. 20, 2015).

[24] Luigi Byun (zziuni@gmail.com), *STUN server list.* [Online]. Available: https://gist.github.com/zziuni/3741933 (visited on Feb. 20, 2015).

[25] Kurento Technologies, *Kurento Open Source Software WebRTC media server.* [Online]. Available: http://www.kurento.org/whats-kurento (visited on Feb. 20, 2015).

[26] M. Cantelon, M. Harter, T. J. Holowaychuk, and N. Rajlich, *Node.js in action.* Greenwich, Conn.: Manning, 2013, ISBN: 9781617290572.

[27] Kurento, *Kurento JavaScript Client JSDoc.* [Online]. Available: http://www.kurento.org/docs/5.1.0/ (visited on Feb. 22, 2015).

[28] O. Etzion and P. Niblett, *Event processing in action.* Greenwich: Manning, 2011, ISBN: 1935182218.

[29] K. Kowal, *Q - a tool for creating and composing asynchronous promises in javascript.*

[30] M. J. Orendorff, *Ecmascript language specification ecma-262 6th edition - draft.*

[31] T. Hughes-Croucher, M. Wilson, *Node: Up and Running: Scalable Server-Side Code with JavaScript*, 1. edition. Sebastopol, CA: O'Reilly & Associates, Apr. 2012, ISBN: 1449398588.

[RFC 6455] I. Fette and A. Melnikov, *The WebSocket Protocol*, RFC 6455 (Proposed Standard), Internet Engineering Task Force, Dec. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6455.txt.

[32] R. Rai, *Socket.io Real-time Web Application Development.* Birmingham: Packt Pub., Feb. 2013, ISBN: 9781782160786.

[33] M. Pol, *Network emulator toolkit.*

[34] Wikipedia, *3g.* [Online]. Available: http://en.wikipedia.org/wiki/3G#Data_rates (visited on Feb. 22, 2015).

[35] ——, *4g.* [Online]. Available: http://en.wikipedia.org/wiki/4G#LTE_Advanced (visited on Feb. 22, 2015).

[36] Kurento, *Kurento JavaScript Client JSDoc*. [Online]. Available: `http://www.kurento.org/docs/current/langdoc/jsdoc/kurento-client-js/index.html` (visited on Feb. 22, 2015).