UNIVERSIDAD CARLOS III DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

TELECOMMUNICATIONS ENGINEERING
FINAL THESIS

Inter-domain Interoperability Framework based on WebRTC

AUTHOR: MIGUEL SEIJO SIMÓ
TUTOR: DR. JOSÉ IGNACIO MORENO NOVELLA

April, 2015

Título:	*INTER-DOMAIN INTEROPERABILITY FRAMEWORK BASED ON WEBRTC.*
Autor:	MIGUEL SEIJO SIMÓ
Tutor:	DR. JOSÉ IGNACIO MORENO NOVELLA

La defensa del presente Proyecto Fin de Carrera se realizó el día 15 de Abril de 2015; siendo calificada por el siguiente tribunal:

Presidente:

Secretario

Vocal

Habiendo obtenido la siguiente calificación:

Calificación:

Presidente **Secretario** **Vocal**

Abstract

Nowadays, the communications paradigm is changing with the convergence of communication services to a model based on IP networks. Applications such as messaging or voice over IP are increasing its popularity and Communication Service Providers are focusing on offering this kind of services.

Moreover, Web Real Time Communication (WebRTC) has emerged as a technology that eases the creation of web applications featuring Real-Time Communications over IP networks without the need to develop and install any plug-in. It lacks of specifications in the control plane, leaving the possibility to use WebRTC over tailored web signalling solutions or legacy networks such as IP Multimedia Subsystem (IMS). This technology brings a wide range of possibilities for web developers, but Communication Service Providers are adviced to develop solutions based on the WebRTC technology as described in the Eurescom Study P2252 [1].

The lack of WebRTC specifications on the signalling platform together with the threats and opportunities that this technology represents for Communication Service Providers, makes evident the need of research on interoperability solutions for the different kind of signalling implementations and experimentation on the best way for Communication Service Providers to obtain the maximum benefit from WebRTC technology.

The main goal of this thesis is precisely to develop a WebRTC interoperability framework and perform experiments on whether the Communication Service Providers should use their existing IMS solutions or develop tailored web signalling platforms for WebRTC deployments.

In particular, the work developed in this thesis was completed under the framework of the Webrtc interOperability tested in coNtradictive DEployment scenaRios (WONDER) experimentation for the OpenLab project. OpenLab is a Large-scale integrating project (IP) and is part of the European Union Framework Programme 7 for Research and Development (FP7) addressing the work programme topic Future Internet Research and Experimentation.

Resumen

Actualmente, el paradigma de comunicaciones está cambiando gracias a la convergencia de los servicios de comunicaciones hacia un modelo basado en redes IP. Aplicaciones tales como la mensagería y la voz sobre IP están creciendo en popularidad mientras los proveedores de servicios de comunicaciones se centran en ofrecer este tipo de servicios basados en redes IP.

Por otra parte, la tecnología WebRTC ha surgido para facilitar la creación de aplicaciones web que incluyan comunicaciones en tiempo real sobre redes IP sin la necesidad de desarrollar o instalar ningún complemento. Esta tecnología no especifica los protocolos o sistemas a utilizar en el plano de control, dejando a los desarrolladores la posibilidad de usar WebRTC sobre soluciones de señalización web específicas o utilizar las redes de señalización existentes, tales como IMS. WebRTC abre un gran abanico de posibilidades a los desarrolladores web, aunque también se recomienda a los proveedores de servicios de comunicaciones que desarrollen soluciones basadas en WebRTC como se describe en el estudio P2252 de Eurescom [1].

La falta de especificaciones en el plano de señalización junto a las oportunidades y amenazas que WebRTC representa para los proveedores de servicios de comunicaciones, hacen evidente la necesidad de investigar soluciones de interoperabilidad para las distintas implementaciones de las plataformas de señalización y de experimentar cómo los proveedores de servicios de comunicaciones pueden obtener el máximo provecho de la tecnología WebRTC.

El objetivo principal de este Proyecto Fin de Carrera es desarrollar un marco de interoperabilidad para WebRTC y realizar experimentos que permitan determinar bajo qué condiciones los proveedores de servicios de comunicaciones deben utilizar las plataformas de señalización existentes (en este caso IMS) o desarrollar plataformas de señalización a medida basadas en tecnologías web para sus despliegues de WebRTC.

En particular, el trabajo realizado en este Proyecto Fin de Carrera se llevó a cabo bajo el marco del proyecto WONDER para el programa OpenLab. OpenLab es un proyecto de integración a gran escala en el cual se desarrollan investigaciones y experimentos en el ámbito del futuro Internet y que forma parte del programa FP7 de la Unión Europea.

Acknowledgements

I would like to thank my tutor, Dr. José Ignacio Moreno, for his guidance and for giving me the opportunity of doing this thesis in the framework of an internship that became such a great experience in my life.

Thanks to all the ”WONDER” team: Steffen, Kay, Paulo, Vasco and Luis. Special mention to my supervisor Steffen Drüsedow, who I thank for his advice, ideas and for trusting me and motivating me to give the best out of myself. Another special mention to Vasco who, even though we worked in different countries, sometimes it felt like we were in the same office. Thanks for your hard work and our great conversations.

Thanks to my internship mates, specially to Arancha and Fran for the amazing time we spent together. I will never forget those days of laughs, ”kicker”, coffee and cake.

I would like to show my appreciation to all the people I met during my Erasmus in Göteborg and my internship in Berlin who became like family to me and I will always remember. I will always treasure the memories of our moments together.

Thanks to my friends, colleagues, lab partners, everyone that was there and helped me during all these years. Thanks to David for so many hours together coding in the basement and for his willingness to work even on Saturday mornings. Thanks also to Grego for his support and cheering me up to give my best in these last steps.

And last, but not least, I will forever be grateful to my parents, José Manuel and Margarita, for their support no matter what. None of this would have been possible without your help. I would like also to mention my brother, Alberto. I hope this will inspire you somehow to follow your goals in life.

Miguel Seijo Simó, Madrid, April 8, 2015

Contents

Glossary **v**

1 Introduction **1**

1.1 Thesis Motivation 1

1.2 Thesis Objectives 2

1.3 Thesis Organization 3

2 State of the Art **5**

2.1 WebRTC 5

2.1.1 WebRTC Architecture and APIs 6

2.1.2 Signalling 7

2.1.3 Network Connectivity 9

2.2 IP Multimedia Subsystem (IMS) 10

2.2.1 Architecture 10

2.2.2 Protocols 15

2.3 Overview of Related Technologies 16

2.3.1 JavaScript 16

2.3.2 JSON 17

2.3.3 HTML5 19

2.3.4 NodeJS 20

2.3.5 WebSockets 21

2.4 Overview of Related Initiatives 22

2.4.1 Telepresence 22

2.4.2 Kurento 22

2.4.3 ORCAjs 23

2.4.4 OpenTok 24

2.4.5 WebRTC2SIP 24
2.4.6 WONDER 26

3 Inter-domain Interoperability Framework based on WebRTC 29
3.1 Scope 29
3.2 Framework description 30
3.3 Application domains 30
3.3.1 Domains description 30
3.4 Interoperability Mechanisms 31
3.4.1 Messaging Server (MS) 32
3.4.2 Client Manager (CM) 34
3.4.3 Messaging Stub 34
3.4.4 Identity Provider 34
3.4.5 Library downloading mechanisms 34
3.4.6 Messaging format 35
3.4.7 Media proxies 35
3.4.8 Trickling support 36

4 System Architecture and Topologies 37
4.1 Functional Architecture 37
4.2 Network Topologies 38
4.2.1 2 party conversations 39
4.2.2 Multiparty conversations 41

5 Application API 45
5.1 Entities 45
5.2 Specification 47
5.2.1 Core Layer 48
5.2.2 Conversation Layer 54
5.3 Development of applications based on the API 63
5.4 Messaging Stub Development 65
5.4.1 Naming convention for a MessagingStub 66

6 Validation 67
6.1 Test environments 67
6.1.1 Deutsche Telekom's Web Centric Test Environment 67
6.1.2 Deutsche Telekom's IMS Test Environment 68
6.1.3 Portugal Telecom's Web Centric Test Environment 70
6.1.4 Portugal Telecom's SIP Centric Test Environment 70

6.2 Results . . . 71
6.2.1 Intra-domain Experimentations . . . 71
6.2.2 Inter-domain Experimentations . . . 73
6.3 Evaluation . . . 76
6.3.1 Interoperability . . . 76
6.3.2 Service delivery (Web centric vs IMS centric) . . . 76

7 Conclusions and further work 79
7.1 Conclusions . . . 79
7.2 Further work . . . 80

Bibliography 86

A Budget 87
A.1 Material Resources . . . 87
A.2 Project Phases . . . 87
A.2.1 Analysis of the State of the Art . . . 88
A.2.2 Experimentation Phase 1: Basic Interoperability, Rich Conversations . . . 88
A.2.3 Experimentation Phase 2: Multiparty, Identities, Resources . . . 89
A.2.4 Documentation . . . 90
A.3 Material Expenses . . . 90
A.4 Human Resources Expenses . . . 90
A.5 Total Expenses . . . 91

B Call Establishment Algorithms 93
B.1 Call Establishment Algorithm for Multiparty . . . 93
B.2 Alternative Call Establishment Algorithm for Multiparty . . . 98

C Code Examples 103
C.1 Example app using the Conversation Layer . . . 103
C.1.1 HTML Code . . . 103
C.1.2 JavaScript Logic . . . 104
C.2 Example app using the Core Layer . . . 106
C.3 Messaging Stub Development . . . 109
C.3.1 Connect() Method . . . 109
C.3.2 SendMessage() Method . . . 110
C.3.3 Disconnect() Method . . . 111

Glossary

API Application Programming Interface, specifies how some software components should interact with each other.. 3–7, 13, 17, 19, 21–23, 29, 30, 33, 34, 45, 47, 54, 63–65, 68, 69, 76, 79, 88–90, 103

AS Application Server. 13, 33, 35, 43, 69

CM Client Manager. 31, 34, 38–40, 68, 69, 93, 98

DT Deutsche Telekom. 26, 29, 30, 32–34, 43, 67–69, 72, 74

eXtensive Markup Language XML. 17

ICE Internet Connectivity Establishment. 6, 7, 9, 10, 22, 25, 36, 52, 53, 58

IDP Identity Provider. 30, 31, 34, 48, 68, 80, 93, 95, 98–100

IMS IP Multimedia Subsystem. 2, 3, 5, 7, 10–15, 22, 26, 30, 31, 33–36, 38, 43, 67–70, 72–74, 76, 77, 79, 80

IMSCM IMS Client manager. 34

JSEP JavaScript Session Establishment Protocol. 8

M. Stub Messaging Stub. 30, 31, 33–35, 37, 39–41, 48, 49, 55, 56, 65, 66, 69, 80, 93, 95, 99, 100

MCU Multipoint Control Unit. 22, 38

MS Messaging Server. vii, 31–36, 39–41, 45, 46, 48–50, 65, 66, 68, 69, 72, 73, 81, 93–96, 98–101

MTI Mandatory to implement. 35

NAT Network Address Translation. 7, 9, 13

NNI Network to Network Interface. 79

OTT Over The Top. 5, 80

PT Portugal Telecom. 26, 29, 30, 33, 67, 69, 70, 72, 74

RCS/Joyn Rich Communication Services/Joyn. 80

SDP Session Description Protocol. 8, 9, 14, 53, 93–96, 98–100

STUN Session Traversal Utilities for NAT. 10, 22

TURN Traversal Using Relays around NAT. 10, 22

WebRTC Web Real Time Communication. vii, 2–9, 17, 19, 20, 22–26, 29–31, 33, 35–38, 45, 46, 54, 55, 57, 58, 61, 63, 64, 72, 76, 79, 80, 88, 106

WONDER Webrtc interOperability tested in coNtradictive DEployment scenaRios. 2, 5, 7, 22, 26, 29, 30, 33, 67, 87

List of Figures

2.1 WebRTC Architecture [2] 6
2.2 WebRTC JSEP Architecture [3] 8
2.3 IMS 3-layer Architecture 11
2.4 JSON Array [4] 18
2.5 JSON Object [4] 18
2.6 NodeJS vs Apache concurrency benchmark [5] 21
2.7 Kurento capabilities [6] 23
2.8 Orca.js Architecture [7] 24
2.9 WebRTC2SIP SIP Proxy architecture [8] 25
2.10 WebRTC2SIP RTCWeb Breaker architecture [8] 25
2.11 WebRTC2SIP Media Coder architecture [8] 26
2.12 WebRTC2SIP Click-to-Call Components [8] 26

3.1 Interoperability mechanisms 31
3.2 Message class 35

4.1 Functional architecture 38
4.2 Proposed Messaging Server (MS) topologies: Caller connects to callee MS 39
4.3 Proposed MS topologies: MS communication 40
4.4 Proposed MS topologies: Both peers connect to remote MS 41
4.5 Mesh multiparty with hosting topology 42
4.6 Multiparty with no hosting topology 42
4.7 Media Stream Star topology 43

5.1 Main API Classes 46
5.2 Main API Classes overview and dependencies 47
5.3 Different API abstraction layers. 64

6.1 DT Web Centric Test Environment 68
6.2 DT IMS Centric Intradomain Test Environment 69
6.3 DT IMS Centric - PT Web Centric Interdomain Test Environment 69
6.4 OSIMS - Open Source IMS experimentation platform 70
6.5 PT Web Centric Intradomain Test Environment 71
6.6 PT SIP Centric Intradomain Test Environment 71

A.1 Gannt diagram 92

B.1 Call establishment algorithm, step 1 - Multiparty with hosting 94
B.2 Call establishment algorithm, step 2 - Multiparty with hosting 94
B.3 Call establishment algorithm, step 3 - Multiparty with hosting 95
B.4 Call establishment algorithm, step 4 - Multiparty with hosting 95
B.5 Call establishment algorithm, step 5 - Multiparty with hosting 96
B.6 Call establishment algorithm, step 6 - Multiparty with hosting 97
B.7 Call establishment algorithm, step 7 - Multiparty with hosting 97
B.8 Call establishment algorithm, step 1 - Multiparty with hosting 98
B.9 Call establishment algorithm, step 2 - Multiparty with hosting 99
B.10 Call establishment algorithm, step 3 - Multiparty with hosting 99
B.11 Call establishment algorithm, step 4 - Multiparty with hosting 100
B.12 Call establishment algorithm, step 5 - Multiparty with hosting 101
B.13 Call establishment algorithm, step 6 - Multiparty with hosting 101
B.14 Call establishment algorithm, step 7 - Multiparty with hosting 102

List of Tables

6.1 Summary of Intra-domain experimentations. Legend in Table 6.9 73
6.2 Two party Inter-domain Basic A/V experimentations. Legend in Table 6.9 . . . 74
6.3 Two Party Inter-domain AV plus Chat Conversation experimentations. Legend in Table 6.9 74
6.4 Two Party Inter-domain Rich Conversation experimentations. Legend in Table 6.9 75
6.5 Multi-party Inter-domain Basic AV experimentations in Mesh Topology. Legend in Table 6.9 75
6.6 Multi-party Inter-domain Basic AV experimentations in MCU based Topology. Legend in Table 6.9 75
6.7 Multi-party Inter-domain Rich Conversation experimentations in Mesh Topology. Legend in Table 6.9 75
6.8 Multi-party Inter-domain Rich Conversation experimentations in MCU based Topology. Legend in Table 6.9 76
6.9 Legend for Tables 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7 and 6.8. 76

A.1 Material expenses 91
A.2 Human Resources expenses 91
A.3 Total Expenses 91

Chapter 1

Introduction

This chapter introduces the motivation, objectives and organization of the thesis for a better understanding of the topic and the document itself.

1.1 Thesis Motivation

During the last decades, the paradigm of communications has been evolving from services with different infrastructures (fixed telephony, mobile telephony, videoconference, TV & video broadcasting, data...) to a set of services based on the Internet (Voice over IP, Video on Demand, videoconference, chat...)[9][10]. Services like Skype or Google Hangouts, that offer chat, voice and videoconference services over Internet are very popular nowadays, but the need of installing a client or a plugin to use the service, the lack of flexibility to integrate these services in different applications and the incompatibility between all these implementations, make a barrier to the adoption of such technologies as an alternative to traditional communication services.

If we focus on Internet-based services, they also suffered a paradigm shift where applications and services are not local anymore, but web-based [11]. Most of the commonly used applications (e.g. text editors, media players, chat applications, etc) can be accessed now online via browser and the data is stored and accessed from the cloud. This change makes applications easier to scale and manage by the service providers and developers, and more convenient and easy to use for the users.

Not only the location, but also the architecture of the Internet-based services changed in the past years. Previously, the client/server architecture was created for a use case where the user was mainly consuming information and media from servers; but the impossibility of scaling servers for the large amount of information that is meant to be shared between users, and the need of reducing latency between user to user communications introduced a new kind of architecture where peers are connected directly (peer-to-peer) and servers are usually used for the signalling and control of such communications. This kind of architecture is specially useful for videoconference, games and file sharing.

Together, all this changes result in the development of WebRTC, a technology that enables Real-Time Communications to run from the cloud in any web browser without the need to develop and install any plug-in. These Real-Time Communications (e.g. conferencing or gaming) are established peer-to-peer for fastest, low latency and more secure connections. The experts conceive WebRTC as a disruptive technology in the telecommunications industry simplifying the task of creating and providing a rich, real-time communications experience as it is to create a web page [12] [13]. WebRTC allows developers to achieve this task by bypassing the traditional network control plane, enabling web technologies to act as a control plane between two users. However, WebRTC can also connect users in legacy networks, enjoying the advantages of Telco operated services like secure authentication, identity management and guaranteed Quality of Service. For IMS legacy networks, 3GPP specification of IMS gives some guidelines and specifications about the support of WebRTC IMS client access to IMS infrastructure. [14]

Not only application developers, but also Communication Service Providers are advised to make use of the WebRTC technology as described in the Eurescom Study P2252 [1], which characterizes WebRTC threats and opportunities for these providers and presents advices on how they could take advantage of this technology.

The fact that signalling is not specified in the WebRTC standard, together with the threats and opportunities that WebRTC represents for Communication Service Providers, makes evident the need of an element filling the gap that this new technology leaves in the control plane, and experimentations on the best way for these Service Providers to obtain the maximum benefit from WebRTC technology.

In particular, the work developed in this thesis was completed under the framework of the WONDER experimentation for the OpenLab project. OpenLab is a Large-scale integrating project (IP) and is part of the European Union Framework Programme 7 for Research and Development (FP7) addressing the work programme topic Future Internet Research and Experimentation.

1.2 Thesis Objectives

Taking into account the situation previously described, the main objective of the work developed in this thesis is to perform practical experiments and evaluate whether Communication Service Providers should use IMS as the control plane for WebRTC services or to use a pure Web based control plane to deliver WebRTC services to their users.

Going more into detail, the main objectives of this thesis are:

- To analyze the state of the art of WebRTC, its related technologies, different signalling system alternatives and the interoperability mechanisms between them.
- To achieve signalling interoperability between different WebRTC service domains. Signalling protocols are not standardized in WebRTC, making this a main challenge for interoperability.
- To create an architecture that makes possible to address peer users that reside in another WebRTC service domain.
- To develop an API for JavaScript based on WebRTC that allows to create clients for any of the interoperable domains making it simple and WebRTC agnostic.
- To create a web-based domain and its connectors to make it work in the interoperable system previously developed.
- To validate the interoperability framework and test the compatibility and functionality in several IMS and web-based domains to clarify which approach is most suitable under which conditions.
- To document the work needed to achieve the previous objectives and the conclusions extracted from this work.

1.3 Thesis Organization

This document's content is organized as follows:

- **Chapter 2: State of the art**

  In this chapter the state of the art is analyzed, starting with WebRTC for a better understanding of the technology used as a base in this work. Later on, the main signalling platforms, IMS and Web Communication Systems, are introduced to have a clear view of its features before evaluating the performance of both platforms during the experimentations. Next section of this chapter gives an overview of related technologies needed to develop this work, while the last section gives an overview of related initiatives to understand the related work performed in the field.

- **Chapter 3: Inter-domain Interoperability Framework based on WebRTC**

  The main contribution on this chapter is on describing the signalling interoperability framework and the web domain developed for this thesis. It is introduced with the scope and a brief description of the framework, followed by a description of all the application domains, including the one developed for this thesis. The last section explains in depth the interoperability mechanisms that will allow to overcome the incompatibilities between different signalling platforms.

- **Chapter 4: System Architecture and Topologies**

  This chapter introduces the elements of the architecture that allow to address peers residing in different domains. It also presents the topologies for the different use cases, to fully understand the way elements interact with each other in order to establish real-time communication between peers.

- **Chapter 5: Application API**

  This chapter explains the API developed for JavaScript and based on WebRTC that allows to create clients for any of the interoperable domains in the most simple and WebRTC agnostic way. Starting with the entities that form this API, the specification section introduces the concept of different layers of the API that allow to create simple and generic or tailored and more complex applications depending on the needs of the developer. Last sections show how to adapt an existing domain to this platform and how to develop an application based on the API

- **Chapter 6: Validation**

  This chapter presents the results obtained from the validation of the framework. First section describes the tests performed and the last sections describe and evaluate the results obtained from these tests.

- **Chapter 7: Conclusions and further work**

  In this chapter, the conclusions from the work previously presented are discussed, ending with a set of recommendations for future work.

- **Annex A: Budget**

  This annex describes the material resources and project phases and calculates the budget needed to develop this work based on the time and resources described.

- **Annex B: Call Establishment Algorithms**

  This annex describes the call establishment algorithms implemented in the solution developed and the alternative proposed as further work.

- **Annex C: Code Examples**

  In this annex, code examples are shown to give a better view of the complexity of implementing a solution compatible with the framework developed for this thesis.

Chapter 2

State of the Art

This chapter aims to review the state of the art of the technologies and initiatives related to the work developed in this thesis. First, Web Real Time Communication (WebRTC) will be introduced for a better understanding of how this technology works, why is it a disruptive technology, its importance, the gaps that this technology leaves and its opportunities, which will be the field of study and experimentations of this thesis. After that the main signalling platform, IP Multimedia Subsystem (IMS), is introduced to analyze its features and understand why this work focuses on this platform to perform the experiments that will lead to a performance comparison between IMS and tailored web based solutions. Next section of this chapter gives an overview of related technologies needed to develop this work, to better understand which technologies are important for the realization of this thesis and why. Finally, the last section gives an overview of related initiatives to understand the related work performed in the field.

The review of the sections mentioned above reveals the importance of WebRTC and highlights the large number of initiatives and projects that have emerged around such a novel technology, predicting a great success for it.

2.1 WebRTC

WebRTC is a technology that brings real time multimedia communications to the web browser without the need of any plugins. It features a JavaScript framework that, together with HTML5 audio and video elements allows web developers to easily create applications that include Peer-to-Peer communications without any additional plugin, lowering the entry barrier to provide Over The Top (OTT) communication services. WebRTC is standardized on a API level at the W3C and at the protocol level at the IETF. [2] [15] [16]

The WebRTC JavaScript API includes the following features for communications reliability:

- Audio and Video codec negotiation

- NAT traversal using Internet Connectivity Establishment (ICE)
- establishment of audio, video and data streams between browsers
- media stream support including noise reduction, echo cancellation and jitter buffers
- a set of mandatory codecs that all implementations should support for guaranteed interoperability

2.1.1 WebRTC Architecture and APIs

To fully understand the role of WebRTC in this work and draw a line around the gaps that this thesis had to cover, it is necessary to have an overview of the WebRTC architecture and API.

The Figure 2.1 shows WebRTC's architecture where there are two interaction layers: (1) WebRTC C++ API and the capture / render hooks, which are interesting for Browser developers; and (2) Web API for application developers, being the latter the one used in the work developed in this thesis.

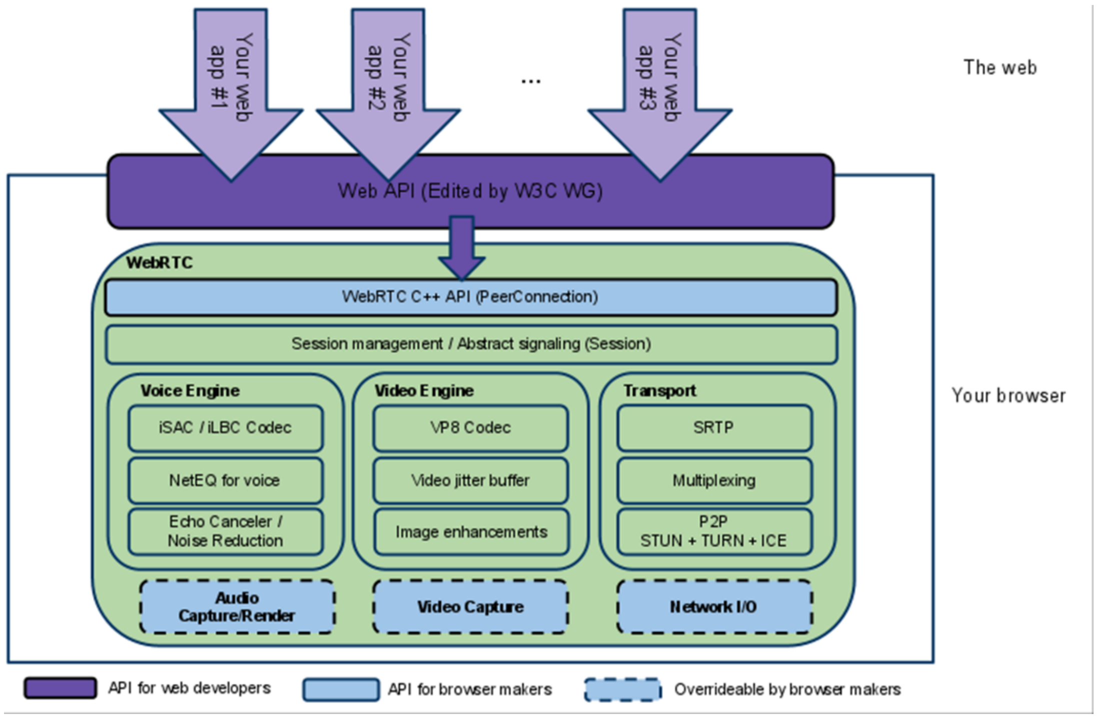


Figure 2.1: WebRTC Architecture [2]

For the Web API interaction in WebRTC, there are 3 APIs implemented:

- MediaStream It represents synchronized streams of media which can contain several VideoTracks and AudioTracks with different "label" values. These streams can be attached to

HTML5 video elements via the createObjectURL() method that returns the BLOB URL that points to the MediaStream. This API also allows to retrieve a MediaStream from the user's webcam and microphone.

- RTCPeerConnection It represents the Peer-to-Peer channel that contains the multimedia (MediaStreams) and data (RTCDataChannel) channels between two peers.

  It manages the invite, accept, reject actions, performs the connection, searches and includes the ICE candidates in both ends of the communication and makes it Network Address Translation (NAT) transversal; but it is not in charge of the signalling needed to transfer this connectivity information to the other end.

  The RTCPeerConnection also hides the complexity that real-time communication implies. The codecs and protocols used by WebRTC deal with the problems that real-time communication over unreliable networks face, such as:

  - Echo cancellation
  - Packet loss concealment
  - Noise reduction and suppression
  - Bandwidth adaptativity
  - Dynamic jitter buffering
  - Image enhancements
  - Automatic gain control

- RTCDataChannel

  It enables low latency and high throughput data communication between peers.

  The RTCDataChannel allows multiple simultaneous channels with prioritization, which are specially useful in scenarios like real-time chat, file transfer, Peer-to-Peer networks, remote desktop applications, gaming, etc.

2.1.2 Signalling

WebRTC uses the elements previously described to stream media and data Peer-to-Peer (browser-to-browser), but signalling is needed to coordinate this communication. Signalling is not specified in the WebRTC standard, leaving it open to different implementations of the signalling platform.

The main contribution of this thesis is to fill this gap and provide an interoperability mechanism among the different signalling alternatives, such as IMS/SIP, XMPP, messaging server over WebSockets or any appropriate duplex communication channel.

Signalling is used to exchange three types of information:

- Session control messages: Initialize and close the communication.
- Network configuration: How can the peers reach each other?.
- Media capabilities: Which codecs and resolutions are compatible for each peer.

Session Establishment

The session must be successfully established with the exchange of signalling messages before Peer-to-Peer streaming can begin.

This exchange of information is outlined by JavaScript Session Establishment Protocol (JSEP)[17]. JSEP's architecture simplifies the client as it avoids the need of the browser being stateful, transferring the state management to the server and avoiding problems with signalling on page reloading. Figure 2.2 shows JSEP's architecture, where JSEP only contains the session description for the WebRTC channel, and is the application the one in charge of the aggregation of this session description to other signalling data to transmit it through the signalling channel.

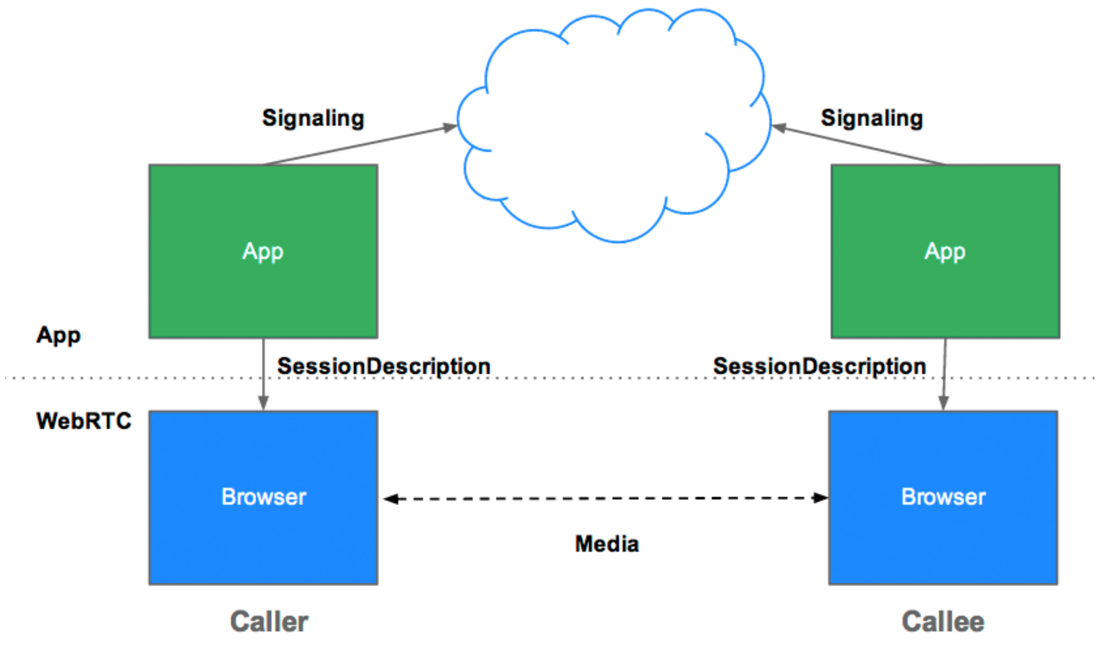


Figure 2.2: WebRTC JSEP Architecture [3]

JSEP defines offers and answers at WebRTC level containing all the signalling information needed for WebRTC to establish a connection. Offers and answers are generated in Session

Description Protocol (SDP) format. Additional signalling information to the included in the SDP could be needed for routing, (e.g. SIP address) or at application level (e.g. the identifiers needed to manage different channels or conversations between peers).

To establish a connection in WebRTC:

1. Alice creates an RTCPeerConnection object.
2. Alice uses the RTCPeerConnection createOffer() method to create an SDP offer.
3. Alice calls setLocalDescription() with the offer created to set her local description.
4. Alice transforms the SDP to a string and uses the signalling mechanisms to send it to Bob.
5. Bob creates an RTCPeerConnection object.
6. Bob calls setRemoteDescription() with Alice's offer to establish Alice's setup in Bob's RTCPeerConnection.
7. Bob calls createAnswer(), that returns Bob's answer with his SDP.
8. Bob calls setLocalDescription() with the answer created to set his local description.
9. Bob transforms the SDP to a string and uses the signalling mechanisms to send it to Alice.
10. Alice sets Bob's answer as the remote session description using setRemoteDescription().

The exchange of connectivity candidates is done in background via the signalling channel as follows:

1. Alice creates an RTCPeerConnection object with an onicecandidate handler.
2. The handler is called when network candidates become available.
3. In the handler, Alice sends stringified candidate data to Bob, via their signaling channel.
4. When Bob gets a candidate message from Alice, she calls addIceCandidate(), to add the candidate to the remote peer description.

2.1.3 Network Connectivity

WebRTC uses Peer-to-Peer connection for real-time communications. This direct connections can be hampered by NAT layers, proxies or corporate firewalls.

WebRTC apps use ICE framework to overcome network complexities. ICE searches the best path to connect peers by testing all the possibilities in parallel and choosing the best option. If the host address is not accessible directly from the other peer, ICE obtains an external address using a Session Traversal Utilities for NAT (STUN) server and in case the connection is not successful, traffic is routed through a Traversal Using Relays around NAT (TURN) relay server.

2.2 IP Multimedia Subsystem (IMS)

IMS is an architectural framework for delivering IP multimedia services. Methods of delivering voice and multimedia services over IP have become popular in the past years (e.g. Skype, Hangouts, VoIP), but not standardized across the industry. IMS is a framework to standardize multimedia IP solutions.

IMS was introduced in the 3rd Generation Partnership Project (3GPP) architecture Release 5 as a solution to offer Internet services everywhere and at any time using cellular technologies [18].

IMS was defined as an architectural framework created for the purpose of delivering IP multimedia services to end-users. It needed to meet the following requirements [19]:

- Support for establishing Multimedia Sessions over packet-switched networks.
- Support for a mechanism to negotiate QoS.
- Support for security.
- Support for interworking with the Internet and circuit-switched networks.
- Support for roaming.
- Support for charging multimedia sessions appropriately.
- Support for strong control imposed by the operator with respect to the services delivered to the end user.
- Support for rapid service creation by standardizing service capabilities instead of services.

As a result, the scope of IMS has been spread and broadened and it aims to be in the heart of convergence between Mobile, Fixed, Broadband and Internet technologies (all-IP networks).

2.2.1 Architecture

First of all, it is important to remark that IMS is responsible for multimedia call control and signalling only. Data itself may be managed by another system (e.g. GPRS/UMTS), although it may be also managed by IMS. Hence, signalling and data flows usually follow a completely different path. Likewise, IMS inherits some concepts from GSM (Global System for Mobile communications) and GPRS (General Packet Radio Service), such as having a home and visited network. In the cellular model, the user is located in the so-called home domain when accesses the network using the infrastructure provided by his own operator. However, if the user roams outside the area of coverage of his home network, he connects through the infrastructure provided not by his operator, but by another one. This infrastructure is called the visited network. Finally, it is also important to keep in mind that 3GPP does not standardize nodes, but functions. For example, several functionalities may be implemented in the same hardware equipment or the same functionality may be distributed among several hardware equipments. Furthermore, these functionalities can be implemented several times along different nodes in a single IMS infrastructure (e.g. for load balancing or organizational issues).

IMS architecture can be split into IMS Core System and IMS Service Framework. As a result, a three layer (i.e. Transport, Control and Service/Application) architecture is obtained (see Figure 2.3).

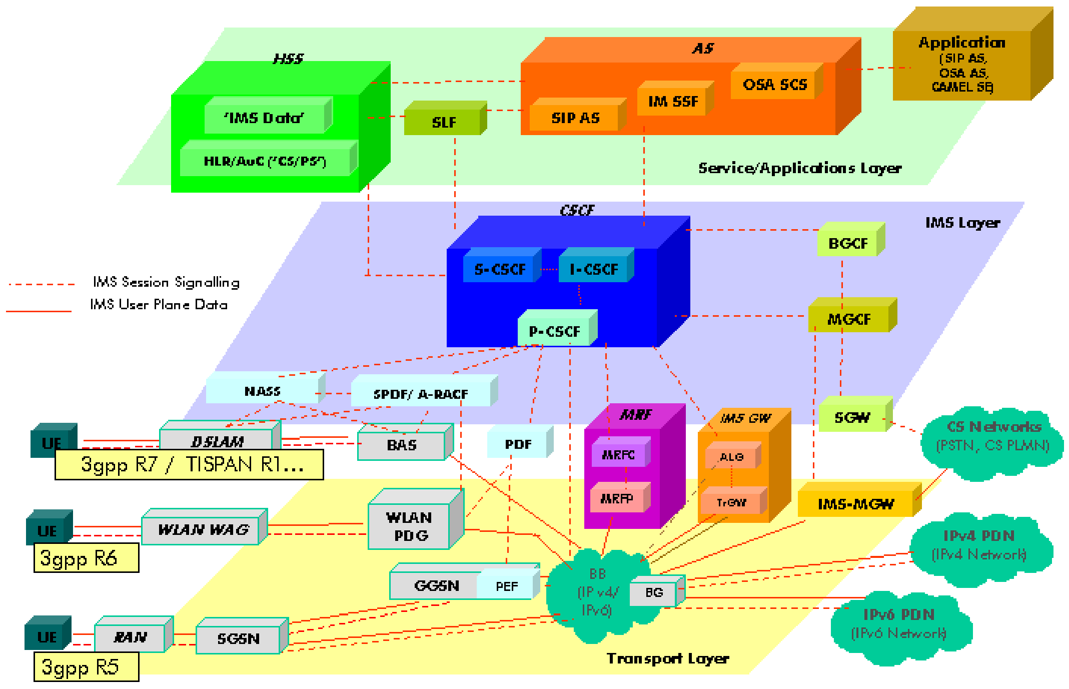


Figure 2.3: IMS 3-layer Architecture

The Databases: Home Subscriber Server and Subscriber Location Function

The Home Subscriber Server (HSS) is the central repository for user-related information. The HSS contains all the user-related subscription data required to handle multimedia sessions. E.g.

location information, security information, user-profile and the Serving-Call/Session Control Function (S-CSCF) allocated to the user. The Subscriber Location Function (SLF) is only needed if there are more than one HSS in the same domain. Basically, it receives a user's address and it returns the HSS where her associated information is stored.

The Call Session Control Function

The Call/Session Control Function (CSCF) is the heart and soul of the IMS. It is a SIP server. It processes SIP signalling in the IMS.

There are three types of CSCF, depending on the functionality they provide:

- P-CSCF (Proxy-CSCF)
- I-CSCF (Interrogating-CSCF)
- S-CSCF (Serving-CSCF)

The P-CSCF is the first point of contact in the signalling plane between the IMS terminal and the IMS network. The P-CSCF is allocated to the IMS terminal during IMS registration. It does not change for the duration of the registration. It can be located either in the home network or in the visited network, being the latter the ideal solution.

The P-CSCF includes several functions:

- Security, e.g. offer integrity protection using IPSec security associations
- Authentication, e.g. assert the identity of the user to the rest of the nodes in the network
- SIP messages handling, e.g. verify the correctness of SIP requests, compress and decompress SIP messages
- Charging
- QoS

Regarding QoS, P-CSCF may include a Policy Decision Function (PDF) or may interface with it (e.g. if it belongs to RACS in the ETSI TISPAN architecture). The PDF authorizes the use of media plane resources and manages QoS over the media plane. The I-CSCF is a SIP proxy located at the edge of an administrative domain. The I-CSCF has an interface, based on the Diameter protocol, to the SLF and the HSS. It retrieves user location information and routes the SIP request to the appropriate destination, typically an S-CSCF. The I-CSCF may optionally encrypt the parts of the SIP messages that contain sensitive information about the domain (THIG, Topology Hiding Inter-network Gateway). The I-CSCF is usually located in the home network, although it may be also located in the visited network, if THIG is put into practice.

From Release 7 onwards, this “entry point” function is removed from the I-CSCF and is located in the Interconnection-Border Control Function (I-BCF) [19]. The I-BCF is used as a gateway to external networks, and it also provides NAT and Firewall functions. The S-CSCF is the central node of the signalling plane. The S-CSCF is essentially a SIP server. But it also performs session control as well as acts as a SIP registrar, i.e. it maintains bindings between the user location and the Public User Identity. It is always located in the home network. The S-CSCF implements a Diameter interface to the HSS. Using this interface, it downloads the authentication vectors of the user who is trying to access the IMS, downloads the user profile, which in turn includes the service profile, and informs the HSS that this is the S-CSCF allocated to the user for the duration of the registration. The S-CSCF handles all the IMS signalling messages associated to a given IMS terminal, decides if one or more application servers are required and finally routes them to the final destination. It also enforces the policy of the network operator.

The Application Server

The Application Server (AS) is a SIP entity (SIP proxy, SIP User Agent or SIP Back to Back User Agent) that hosts and executes services. The S-CSCF may interface with them to fulfil the IMS user’s requirements. Three different types of AS are defined:

SIP AS: this is the native Application Server that hosts and executes IP Multimedia Services based on SIP. It can be located in the home network or in a third party network.

OSA-SCS (Open Service Access-Service Capability Server): this Application Server provides an interface to the OSA framework. It inherits all the OSA capabilities, such as the capability to access the API securely from external networks. This node interfaces, on the one side, the S-CSCF using SIP and, on the other side, the OSA Application Server using the OSA API. It is located in the home network; although OSA ASs can be located either in the home network or in a third party.

IM-SSF (IP Multimedia-Service Switching Function): this Application Server allows IMS to keep on using CAMEL (Customized Applications for Mobile network Enhanced Logic) services that were developed for GSM. The IM-SSF allows a gsmSCF (GSM Service Control Function) to control an IMS session. This node interfaces, on the one side, the S-CSCF using SIP and, on the other side, the gsmSCF using a protocol based on CAP (CAMEL Application Part). It is located in the home network.

By keeping services independent of the IMS ‘standardization’, development of new services is encouraged and the scalability and modularity of the architecture is improved.

The Media Resource Function

The Media Resource Function (MRF) provides a source of media in the home network. The MRF provides the home network with the ability to play announcements, mix media streams, transcode between different codecs, obtain statistics, and do any sort of media analysis. The MRF is further divided into a signalling plane node called the Media Resource Function Controller (MRFC) and a media plane node called the Media Resource Function Processor (MRFP). The MRFC acts as a SIP User Agent to the S-CSCF and it also controls the resources in the MRFP via an H.248 interface. The MRFP implements all media-related functions.

IPv4 -IPv6 interworking: IMS-Application Layer Gateway and Transition Gateway

IMS supports two IP versions, namely IPv4 [20] and IPv6 [21]. At some point in an IP multimedia session, interworking between them may occur. In order to facilitate interworking between IPv4 and IPv6 without requiring terminal support, the IMS adds two new functional entities that provide translation between both protocols. These new entities are the IMS Application Layer Gateway (IMS-ALG), that processes control plane signalling (e.g. SIP and SDP messages), and the Transition Gateway (TrGW), that processes user plane traffic (e.g. RTP, RTCP). The IMS-ALG acts as a SIP Back to Back User Agent by maintaining two independent signalling interfaces: one towards the internal IMS network and the other towards the other network. Each of these interfaces is running over a different IP version. In addition, the IMS-ALG rewrites the SDP by changing the IP addresses and port numbers created by the terminal with one or more IP addresses and port numbers allocated to the TrGW. This allows the user plane traffic to be routed to the TrGW. The IMS-ALG interfaces the I-CSCF for incoming traffic and the S-CSCF for outgoing traffic. The TrGW is effectively a NAT-PT/NAPT-PT (Network Address Port Translator–Protocol Translator). The TrGW is configured with a pool of IPv4 addresses that are dynamically allocated for a given session. The TrGW performs the translation of IPv4 and IPv6 at the media level (e.g. RTP, RTCP).

The interface with the Circuit-Switched Network

In the IMS architecture there are several nodes specially defined to manage the interaction between the IMS and circuit-switched networks, such as the Public Switched Telephone Networks (PSTN).

These nodes are:

The BGCF (Breakout Gateway Control Function) is essentially a SIP server that includes routing functionality based on telephone numbers. Its main task is:

- to select an appropriate network where interworking with the circuit-switched network is to occur.

- or, to select an appropriate PSTN/CS gateway, if interworking is to occur in the same network where the BGCF is located.

The PSTN/CS Gateway provides an interface toward a circuit-switched network, allowing IMS terminal to make and receive calls to and from the PSTN (or any other circuit-switched network). The PSTN/CS Gateway is in turn decomposed into the following functions:

SGW (Signalling Gateway): the SGW interfaces the signalling plane of the CS network performing lower layer protocol conversion (e.g. transform ISUP or BICC over MTP into ISUP or BICC over SCTP/IP1).

MGCF (Media Gateway Control Function): the MGCF is the central node of the PSTN/CS Gateway. The MGCF maps SIP (the call control protocol on the IMS side) to either ISUP over IP or BICC over IP, which are handled by SGW. The MGCF also controls the resources in a MGW using H.248.

MGW (Media Gateway): the MGW interfaces the media plane of the PSTN or CS network. On one side, it is able to send and receive IMS media over RTP. On the other side, it uses one or more PCM (Pulse Code Modulation) time slots to connect to the CS network. Additionally, the MGW performs transcoding when the IMS terminal does not support the codec used by the CS side, e.g. IMS might use AMR (Adaptive Multi-Rate compression) and PSTN might use G.711.

2.2.2 Protocols

IMS is entirely built on Internet protocols defined by the Internet Engineering Task Force (IETF). The most important ones are:

SIP (Session Initiation Protocol) / SDP (Session Description Protocol): used to establish and manage multimedia sessions. SIP works end to end since it does not differentiate the User-to-Network Interface (UNI) from a Network-to-Network Interface (NNI) [22], [23].

Diameter: it is an AAA (Authentication, Authorization and Accounting) protocol. It consists of a base protocol [24] that is complemented with so-called Diameter Applications. E.g. IMS defines a Diameter application to interact with SIP during session setup and another one to perform credit control accounting.

COPS (Common Open Policy Service): it supports policy control over QoS signalling protocol (e.g. Resource ReSerVation Protocol, RSVP). It is used to convey policy requests and decisions between Policy Decision Points (PDPs) and Policy Enforcement Points (PEPs)

[25]. H.248, also known as MEGACO (MEdia GAteway COntrol): H.248 is used to control nodes in the media plane (e.g. a media gateway controller controlling a media gateway) [26].

RTP (Real-Time Transport Protocol) / RTCP (RT Control Protocol): RTP is used to transport real time media, such as audio and video. RTCP does not transport any data itself, but provides feedback on the QoS being provided by RTP [27].

2.3 Overview of Related Technologies

This section will describe other technologies used to develop the work needed for this thesis.

2.3.1 JavaScript

JavaScript is an object-oriented, prototype-based dynamic scripting programming language commonly used to execute programs in web browsers. It has been standardized in the ECMAScript language specification. In the client-side it is used for user interaction, asynchronous communications, and to dynamically modify the website that is displayed. The speed increase on the newer JavaScript virtual machines (VMs) and platforms built upon them, such as Node.js, have increased the popularity of JavaScript for server-side web applications. This language is also used for other than web-based applications, such as PDF documents, desktop widgets, etc. JavaScript is increasingly being used as a compile target for source-to-source compilers, including compilers that allow C and C++ programs to be compiled into JavaScript and execute at near-native speeds, making JavaScript to be considered the ”assembly language of the web” [28]

Use cases

JavaScript's most common use is to add client-side behavior to HTML pages. Scripts are included in the code or imported from HTML pages and interact with their content using the Document Object Model (DOM) of the HTML page. Because JavaScript code can run locally in the browser rather than in the remote server, it makes the applications more responsive and less dependant on the connection. Furthermore, JavaScript can detect user actions and extends the capacities of HTML, such as individual keystrokes.

Most usual client-side use cases for JavaScript are:

- Validate and process data introduced by the user.
- Update the page content and submit data to the server via AJAX without reloading the page.

- Animation and visual processing.
- Games and interactive content.

A JavaScript engine is an interpreter of the JavaScript source code that executes the script accordingly. The first JavaScript engine, called Spider-Monkey, was created for the browser Netscape Navigator and implemented in C.

The work done in this thesis is based on WebRTC to support multimedia services on the browser, whose main APIs are for JavaScript. This makes JavaScript the most convenient choice as the programming language for the client for its WebRTC compatibility and because it is the reference language for client side in the web applications.

2.3.2 JSON

JSON is an open standard data format that transmits objects consisting of attribute-value pairs using human-readable text. It is the main alternative to XML (eXtensive Markup Language) to transmit data between server and web applications. The JSON format was originally specified by Douglas Crockford, and described in RFC 4627 [29] and ECMA-404. Even if JSON is derived from JavaScript, it is a language-independent data format. Libraries for parsing and generating JSON data are available for a large number of programming languages. JSON's design goals were to make it minimal, portable, textual, and a subset of JavaScript.

In general, the main advantages of JSON are:

- Language independent.
- Easy to read and write for humans and to parse and generate for the machines.
- Uses conventions that are familiar to programmers.
- Data-oriented and can be mapped easily to object-oriented systems.

JSON is built on two universal data structures which all modern programming languages can support somehow.

JSON's data structures can be:

- A collection of name/value pairs. This can be realized as an object, record, struct, dictionary, hash table, keyed list, or associative array in the different programming languages.
- An ordered list of values. Which is equivalent to an array, vector, list, or sequence in the different programming languages.

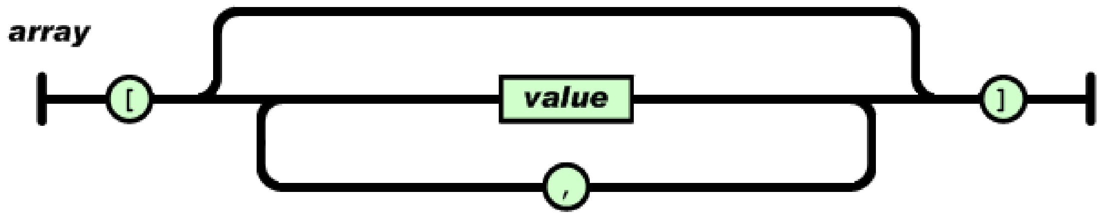


Figure 2.4: JSON Array [4]

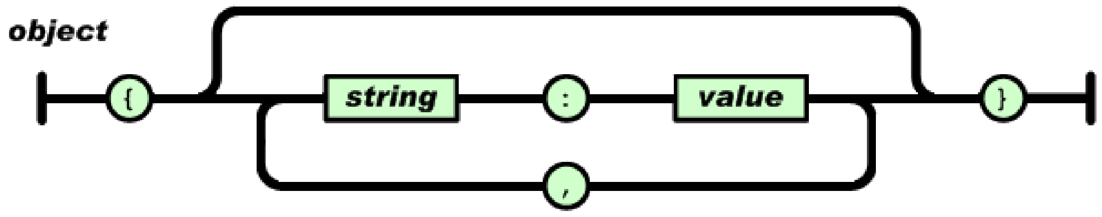


Figure 2.5: JSON Object [4]

The values inside an object or array can be a string, an object, an array, a boolean (true or false) or a null.

The following example shows a possible JSON representation describing an entry in a phone agenda.

```
{
  "Name": "Alice Balmer",
  "address": {
    "streetAddress": "Puerta del Sol, 1",
    "city": "Madrid",
    "postalCode": "28001"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "647382910"
    },
    {
      "type": "office",
      "number": "646555898"
    }
  ],
  "emailAddresses": [
    {
      "type": "home",
      "emailAddress": "alice_balmer_80@gmail.com"
    },
    {
      "type": "office",
      "number": "a.balmer@pixmar.com"
    }
  ]

}
```

The simplicity of JSON and the availability of native JSON parser made it the obvious choice as the message exchange format for the proof-of-concept web domain developed for this thesis.

2.3.3 HTML5

HTML5 is the main markup language of the Internet used to present content for the World Wide Web. Finished in 2014, it is the fifth revision of the HTML standard of the World Wide Web Consortium (W3C) [30].

Its main goal is to improve the HTML language supporting new multimedia contents while keeping the readability and ease to parse featured in the previous versions of HTML. HTML5 is also focused on interoperability, including detailed processing models so the implementations are more compatible, featuring APIs for complex web applications [31] and being able to run on low-powered devices such as smartphones.

HTML5 includes the new <video>, <audio> and <canvas> elements, Scalable Vector Graphics (SVG) content, and MathML for mathematical formulas in order to make easier to include and handle multimedia and graphical content. Other elements enhance the semantic content of the HTML5 documents (e.g. <section>, <article>, <header>, <nav>, etc.). The APIs and Document Object Model (DOM) have become fundamental parts of the HTML5 specification.[31]

APIs

For HTML5, the W3C proposed to increase modularity as a key part of the plan to make faster progress, making some technologies originally defined in HTML5 itself now defined in separate APIs, such as:

- WebRTC WG – WebRTC
- Web Apps WG – Web Messaging, Web Workers, Web Storage, WebSocket API, Server-Sent Events

- HTML Working Group – HTML Canvas 2D Context
- IETF HyBi WG – WebSocket Protocol
- W3C Web Media Text Tracks CG – WebVTT

On the contrary, also some initially standalone specifications have been adapted as HTML5 extensions or features, like: SVG, MathML, WAI-ARIA.

In this project, HTML5 is the base of the client frontend, being of special importance its WebRTC and WebSockets capabilities.

2.3.4 NodeJS

Node.js is a software platform designed to build scalable applications (especially server-side) using JavaScript. It achieves high throughput by using non-blocking I/O and a single-threaded event loop.

Node.js features a built-in HTTP server library to make applications able to easily run a web server and create web applications. Node.js scales better than a typical Linux-Apache-MySQL-PHP (LAMP) server stack, as in a LAMP server stack each new connection to the server spawns a new thread. This works well with few connections, but as the number of users increases it loses performance, being adding more servers the only way to support a large number of users. The 2.6 shows how the performance is reduced for a large number of concurrent connections.

In Node.js each new connection triggers an event, which means that Node.js will never lock up, supporting a large number of concurrent users. In theory, Node.js can handle as many connections as the maximum number of sockets supported by the system (e.g. 65.000 connections for an UNIX system), but in practice the number of connections depends on the amount of information exchanged between server and client among others, handling around 25.000 clients without reducing the performance.

The main disadvantage of Node.js against other server application platforms is that using a single thread only allows it to use one CPU. This limitation can be overcome by starting several instances controlled by a load balancer.

NodeJS was used for the server side part in the proof-of-concept domain developed for this thesis, which will be thoroughly explained in the following chapters of this document.

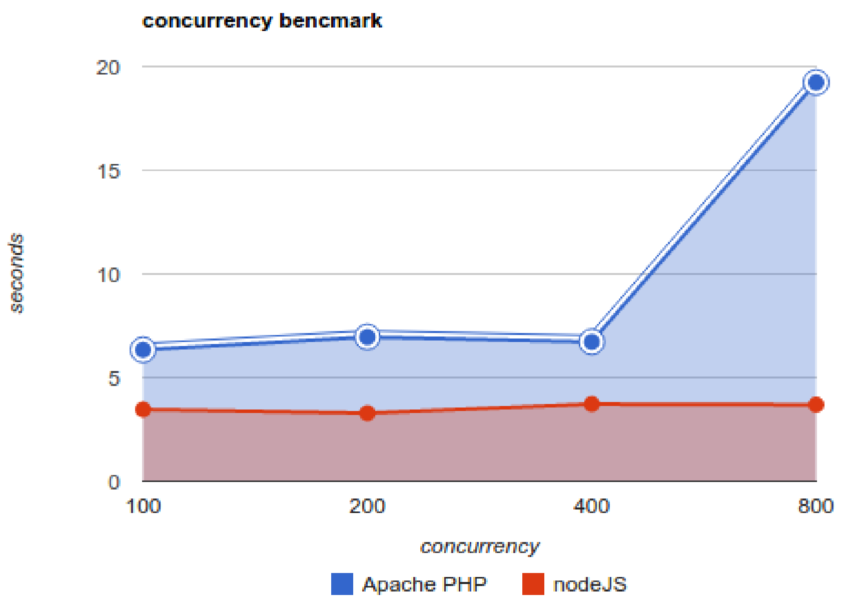


Figure 2.6: NodeJS vs Apache concurrency benchmark [5]

2.3.5 WebSockets

WebSockets technology has been defined in the World Wide Web Consortium (W3C)'s WebSocket API specification. It consists on a JavaScript API and protocol for bidirectional communications over the Internet that simplifies working with fixed data formats and bypasses the slower HTTP protocol.

The current HTTP standard protocol needs to request documents from a server and wait for the document to be sent before it can display a web-page, which makes it slower than WebSockets, where you can send and receive your data immediately using text, binary arrays, or blobs.

The main advantages of WebSockets are the support of duplex communications, its increased client-server communication efficiency, the use of TCP for reliable communications and its speed, being faster than HTTP.

Its main disadvantages are the browser compatibility (it must be fully HTML5 compliant) and the fact that it takes over the communications protocol between the client and the server for a particular connection.

WebSockets are specially useful in cases that need duplex and long-term communication without the need of supporting the request-response process. This is specially useful for loading pages dynamically or any kind of client-server communication.

For the development made for the proof-of-concept web based domain, WebSockets were used mainly for simplicity and convenience as it's the easiest way to develop client-server communication in JavaScript.

2.4 Overview of Related Initiatives

This section will give an overview of other initiatives related to WebRTC technology, such as Multipoint Control Unit (MCU) systems which are compatible with WebRTC like Telepresence and Kurento (that will be needed for the Star Topology described in Section 4.2.2), APIs that help and add value for WebRTC developers like ORCAjs and OpenTOK, SIP compatibility layers like WebRTC2SIP or related projects like Webrtc interOperability tested in coNtradictive DEployment scenaRios (WONDER).

2.4.1 Telepresence

Telepresence [32] is an open source Multipoint Control Unit (MCU) for audio and video mixing that allows SIP clients to create an unlimited number of bridges to establish multimedia conferences between a virtually unlimited number of participants. It offers support for SIP registrar, 4 different protocols to deliver the SIP messages (WebSocket, TCP, TLS and UDP) as well as for WebRTC clients. NAT traversal technologies (i.e. symmetric RTP, RTCP-MUX, ICE, STUN and TURN) allow connectivity even with clients behind firewalls and different networks while PSTN interconnection support allows its compatibility with legacy phone switched networks. Telepresence can mix different audio and video codecs on a single bridge and can be configured as stand alone server or AS (Application Server) to use it behind a server such as Asterisk or an IMS server.

This MCU system was the one used in this project to make experimentations with MCU based topologies, as will be explained in the Tests section.

2.4.2 Kurento

Kurento is an open source WebRTC media server and a set of client APIs to simplify the development of advanced video applications for online and mobile plaftorms. It features transcoding, mixing, recording, group communications, broadcasting and advanced media processing capabilities like computer vision, augmented reality or video indexing, as shown in Figure 2.7.

Its modular architecture allows the integration of third party media processing algorithms with developers' applications.

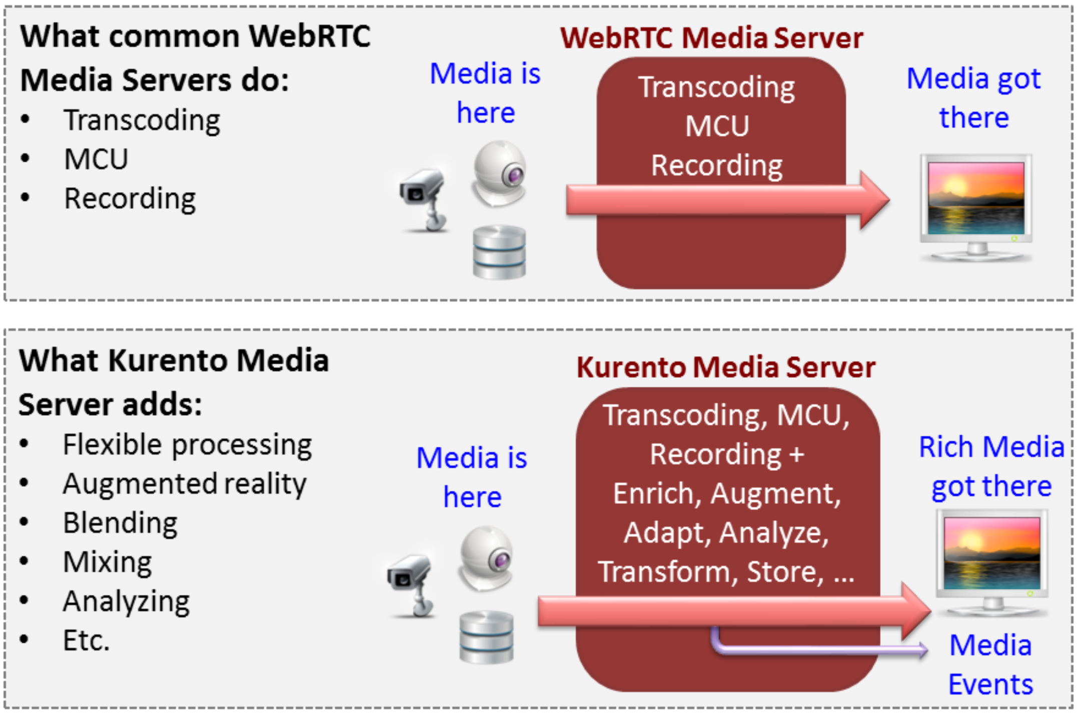


Figure 2.7: Kurento capabilities [6]

2.4.3 ORCAjs

Orca.js software aims to standardize and simplify signalling for WebRTC providing the tools and JavaScript libraries to fill the gap left by WebRTC.

Orca.js is divided in two parts:

Orca.js API Specification Code The Open Source orca.js API only contains the specification and does not implement an operational service. It is intended for application developers that want to inspect the API or developers of transport libraries.

Orca.js Reflector SDK The Open Source orca.js Reflector SDK consists on a simple ORCA service built on node.js for developing and testing applications. It includes an operational transport library and a simple application to test and illustrate how to use the API.

OrcaJS was considered as a base for the API developed for this work, but it was finally discarded because OrcaJS isn't designed for interoperability, doesn't support multiparty conversations nor rich features (e.g. chat, file sharing,...), doesn't support identity management and it is call oriented instead of conversation oriented, being a much more convenient option to create the API from scratch.

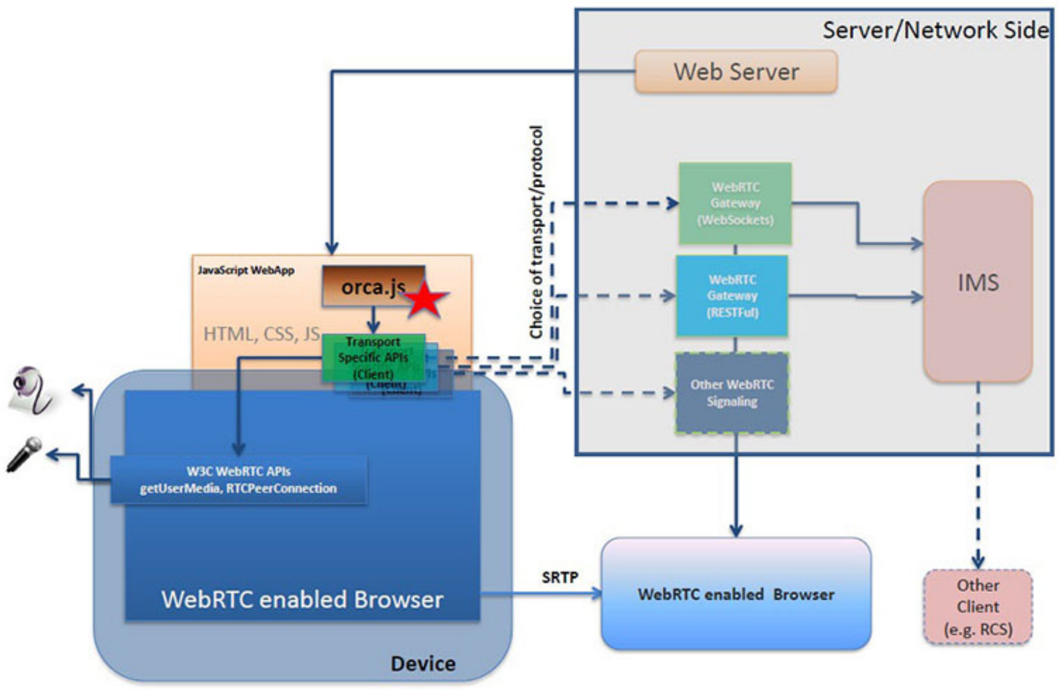


Figure 2.8: Orca.js Architecture [7]

2.4.4 OpenTok

The OpenTok is a WebRTC platform that deals with the hassle of developing, maintaining, and monitoring the infrastructure that is usually associated with the development of a real-time multimedia application.

The OpenTok platform is composed of two parts:

- OpenTok client-side libraries (for JavaScript, iOS, and Android).
- OpenTok server SDKs, (in Java, PHP, Python, Ruby, .NET, Node.js). It provides a simple interface to create video chat sessions and to authenticate users.

OpenTok is a full featured WebRTC platform but it couldn't be used for this project because of the lack of interoperability with other platforms/domains and because its license is not open source so it was not possible to use it as a base for the experimentation.

2.4.5 WebRTC2SIP

WebRTC2SIP is a gateway developed by Doubango Telecom that uses WebRTC and SIP to enable making and receiving calls from/to any SIP-legacy network or PSTN in the browser.

The gateway contains four modules: A SIP Proxy, RTCWeb Breaker, Media Coder and Click-to-Call feature.

SIP Proxy

The SIP Proxy module converts the WebSocket protocol used in the browser to UDP, TCP or TLS which are the protocols supported by legacy SIP servers. It acts as a transparent proxy so there are no special requirements for compatibility in the end server.

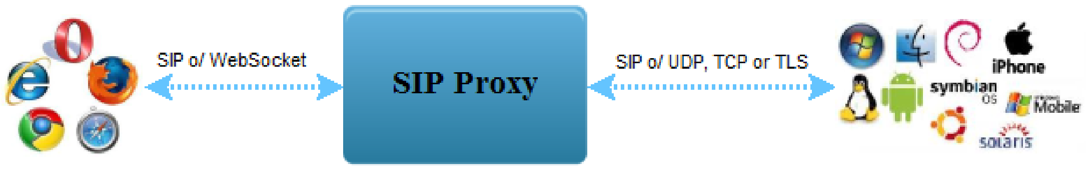


Figure 2.9: WebRTC2SIP SIP Proxy architecture [8]

RTCWeb Breaker

WebRTC specifications include mandatory support for ICE and DTLS/SRTP while many SIP-legacy endpoints (e.g. PSTN network) do not support them. The RTCWeb Breaker negotiates and converts the media stream to allow interoperability.

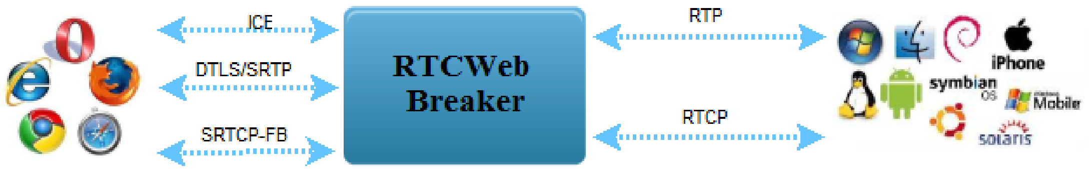


Figure 2.10: WebRTC2SIP RTCWeb Breaker architecture [8]

Media Coder

The WebRTC standard defined two audio codecs which are mandatory to implement: opus and g.711. The video codecs which are mandatory to implement are not chosen yet. The choice is between VP8 and H.264 where VP8 is royalty-free but not widely deployed and H.264 AVC is not free but widely deployed. Each implementation of WebRTC uses one of these codecs, making the need of a media coder to transcode the video streams.

Click-to-Call

More a service than a module, it acts as a SIP click-to-call solution. It allows to send a link via email or post it on a website and allow other people to call you with a single click.

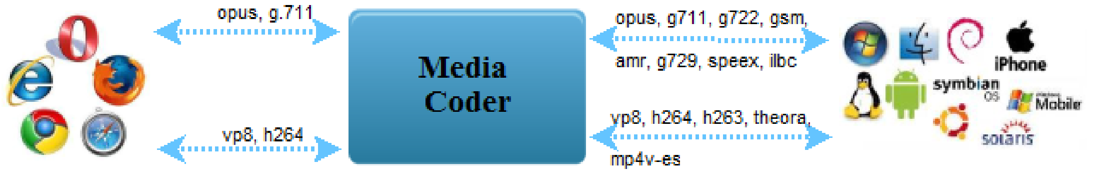


Figure 2.11: WebRTC2SIP Media Coder architecture [8]

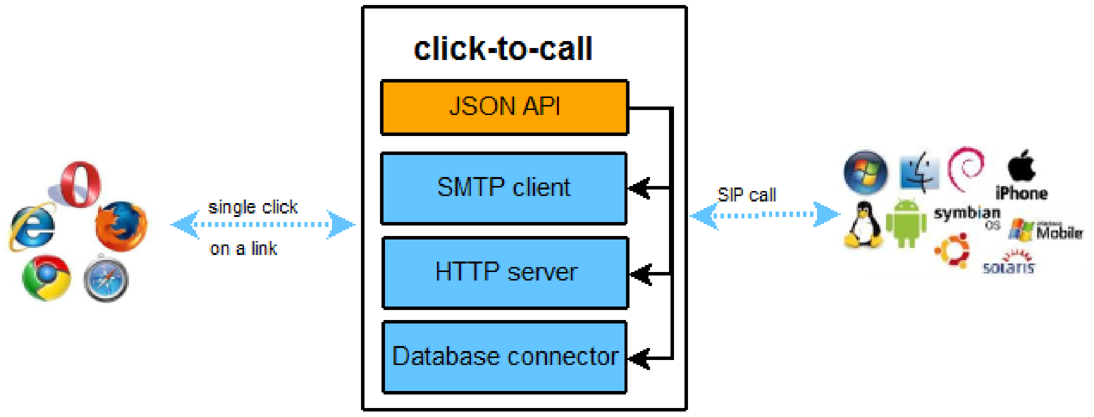


Figure 2.12: WebRTC2SIP Click-to-Call Components [8]

WebRTC2SIP gateway was the base for the compatibility modules for IMS/SIP domains that were developed in the framework of the WONDER project and used for testing and results in this thesis.

2.4.6 WONDER

The Webrtc interOperability tested in coNtradictive DEployment scenaRios (WONDER) project its the result of a partnership between Deutsche Telekom (DT) and Portugal Telecom (PT) that was partially funded by the European Commission as a part of the OpenLab project. OpenLab is a Large-scale integrating project (IP) and is part of the European Union Framework Programme 7 for Research and Development (FP7) addressing the work programme topic Future Internet Research and Experimentation. The reports and deliverables from the WONDER project are public and published on the openlab website [33]

In this project, the main task is to perform experiments with WebRTC services on IMS infrastructures and WebRTC services on domains implemented with pure web technologies. The experiments aim to clarify which approach is most suitable and in which conditions. It also tries to cover how to manage different models of user identities and how to address users that reside in another domain and how to notify users when they receive a call but they are not logged in or have no browser opened.

The work developed in this thesis was part of the WONDER project, where the contribution of this work was mainly focused on the development of the interoperability framework and a simple web domain used for testing.

Chapter 3

Inter-domain Interoperability Framework based on WebRTC

This chapter introduces the contribution made in this thesis to develop an inter-domain interoperability framework based on Web Real Time Communication (WebRTC). The first section summarizes the parts of such an interoperability framework that are covered by this thesis. Next sections describe the framework and introduce the domains that will be tested for a better understanding of the framework and tests. Last section explains the interoperability mechanisms used to achieve compatibility between the different domains, as it is important to fully understand how the framework works.

This first approach to the interoperability framework shows how the use of libraries downloaded on the fly and messaging servers for each specific domain are the key elements for interoperable domains. Next chapters will focus on the architecture and topologies for call establishment and the API which includes the interoperability mechanisms previously mentioned.

3.1 Scope

The work developed in this thesis aims to achieve signalling interoperability between different WebRTC domains. In order to do this, the architecture, the API and the interoperability mechanisms used to achieve compatibility among the different domains were defined in the scope of the Webrtc interOperability tested in coNtradictive DEployment scenaRios (WONDER) project by all the parties involved (i.e. the author of this thesis as part of the Deutsche Telekom (DT) team, and the Portugal Telecom (PT) team). Common parts were implemented along with the API as part of the work developed for this thesis. All the domain specific parts were created and deployed by the partner responsible of each particular domain, where the proof-of-concept Deutsche Telekom (DT) web domain was developed and deployed as part of the work for this thesis.

The tests performed to check the interoperability among the different domains intend to evaluate in which cases is recommended to use IP Multimedia Subsystem (IMS) as the signalling platform for WebRTC services or a pure Web based platform to deliver these services. The tests were performed by the responsible of the domains involved and analyzed in the scope of this thesis.

Note that the evaluation of the WebRTC technology per-se is out of the scope of this work, which focus on the signalling and service delivery for different domains.

3.2 Framework description

The inter-domain interoperability framework uses WebRTC as core for the multimedia communication, where the framework is in charge of the logic of the interoperability mechanisms, the topology used in the communication and the signalling.

The framework includes an API that manages all the logic on the client side for this kind of application, the different servers used for signalling, an Identity Provider (IDP) to resolve and know how to reach users from foreign domains, media proxies in case some media conversion has to be made and a Messaging Stub (M. Stub) for each domain that works as an adapter to the different implementations of the signalling servers.

These elements ensure compatibility among different domains where the different servers use diverse technologies and implementations.

The next sections will introduce the domains where the framework was deployed and tested and the interoperability mechanisms behind all the elements presented above.

3.3 Application domains

The features of the system developed for this thesis were deployed and tested in four different domains, focusing on the interoperability among clients in all of them.

Two of the domains are based on IMS and the other two on pure Web services (where one was developed by the Deutsche Telekom (DT) team and one by the Portugal Telecom (PT) team) The DT web centric domain was developed by the author as part of the DT team in the scope of this thesis while the other domains were developed by DT and Portugal Telecom (PT) under the WONDER project.

3.3.1 Domains description

IMS based service delivery

IMS based Service delivery uses the existing IMS infrastructures to support WebRTC services. This approach is specially useful for communication service providers that already have deployed a full IMS network infrastructure that want to provide WebRTC services using their current infrastructure. Nevertheless, this may incur in additional costs from upgrades in the IMS infrastructure that should be justified with other commercially strong use cases.

Web-based service delivery

Web based Service delivery is mainly a ”tailored” WebRTC service delivery where the implementation of the signalling/service delivery platforms are completely based on Web Technologies. This approach has the web application server as the core network element to provide basic session control and also advanced communication features. The lack of standardization and available support of this platform compared to IMS service delivery, together with the fact that all services should be designed and developed from scratch makes it an attractive option for systems where the operation cost per WebRTC endpoint must remain as low as possible.

3.4 Interoperability Mechanisms

The main challenges for the design of the framework were to make an universal and scalable solution, not only valid for IMS and web, but also to a virtually unlimited number of domains that may use completely different technologies than the ones tested.

Universal interoperability is accomplished with the inclusion of Messaging Server (MS), Client Manager (CM), Messaging Stub (M. Stub), media proxies, unified messaging format and trickling support.

Scalable interoperability is implemented in the creation of an unified Identity Provider (IDP) and dynamic download of the domain specific libraries.

The Figure 3.1 shows how these mechanisms are used.

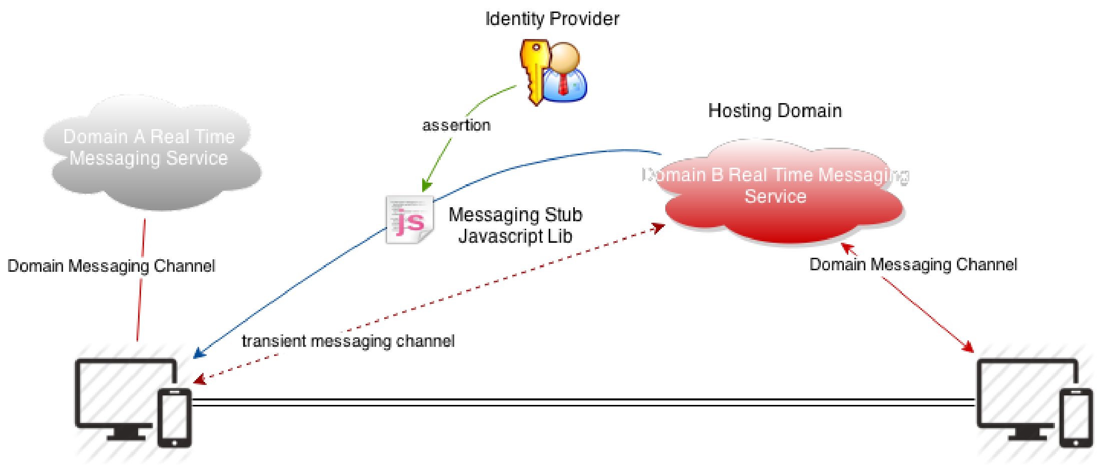

Identity Provider
assertion
Hosting Domain
Domain A Real Time Messaging Service
Domain B Real Time Messaging Service
Messaging Stub Javascript Lib
Domain Messaging Channel
Domain Messaging Channel
transient messaging channel

Figure 3.1: Interoperability mechanisms

3.4.1 Messaging Server (MS)

The Messaging Server (MS) is one of the most important parts used to achieve interoperability between the different domains. It works as a common signalling gateway for clients to deliver the signalling messages to other clients inside a domain. Every domain will have its own MS.

In the next subsections, the different MSs will be explained, focusing on the Web-based DT domain which was developed in the scope of this thesis by its author.

Web-based DT Messaging Server

The Web-based DT MS is a basic Nodejs (see Section 2.3.4) script that uses WebSocket (see Section 2.3.5) technology. The clients can connect and do a basic login, so then they can send messages to other connected clients.

All messages sent/received by this MS follow JSON format (see Section 2.3.2) with the following structure:

Login message:

message = {

type: "login"	The type field of the message will be the reserved word "login"
from: "username@dt-web.de"	The from field of the message will be the username chosen

}

Message sending:

message = {

type: "message"	The type field of the message will be the reserved word "message"
from: ""	The from field of the message will be automatically filled
to: "recipient@dt-web.de"	The to field of the message will be the recipient of the message
body: signalling_message	The body of the message will contain the signalling message

}

From the point of view of implementation, the server will be listening for incoming connections and every time a login message is received, it will map the username with the connection. From that moment on, the rest of the messages received from any connection that have this username as recipient will be redirected to this connection. The message could have followed the messaging format used by the interoperability framework designed for this thesis, but this additional header was implemented to make the basic functionality agnostic to signalling implementations and support changes in the interoperability framework message format without changing the MS.

This basic MS has the minimum features required, allowing to test the feasibility of implementing the extra features in the M. Stub, the API or the need to implement the functionality in the MS itself. For example, the Multicast feature (needed for multiparty conversations) its be supported by the M. Stub.

IMS-based DT Messaging Server

The IMS-based DT MS was developed by DT in the scope of the WONDER project. This server was programmed in Java using a library to support SIP and is registered as an Application Server (AS) in the IMS system. To communicate with the clients, it uses WebSocket technology and JSON notation. It translates between messages formatted with the format designed for interoperability and SIP compliant messages, avoiding the need of a SIP library in the browser, and its designed to allow clients from other domains to send signalling messages to the IMS core, which will redirect the messages to any client in the domain.

Web-based PT Messaging Server

The Web-based PT MS was developed by PT for experimentation in the WONDER project. It also offers a WebSocket-based JSON interface, but unlike the web-based DT MS, this one is based in a more complex Vert.x application platform and programmed in Java. It integrates different modules that communicate each other through an event bus and includes advanced features such as multicast.

SIP-based PT Messaging Server

The SIP-based PT MS was also developed by PT during the WONDER project. In this domain, the switching capabilities are provided by Asterisk with WebRTC2SIP (described in Section 2.4.5) as an interface to compatibly it with WebRTC. The WebRTC specifications include mandatory support for ICE and DTLS/SRTP. The problem is that many SIP-legacy endpoints (i.e. Asterisk) do not support these features. WebRTC2SIP's RTCWeb Breaker module negotiates and converts the media stream to make these two worlds compatible.

Unlike the IMS-based DT MS, in this domain the MS receives SIP messages, leaving the translation from the messaging format used by the interoperability framework to the M. Stub.

3.4.2 Client Manager (CM)

The Client Manager (CM) is the signalling gateway that clients use to connect to their own domain. It adapts the connections, protocols and message format from the framework standards to the ones used by the underlying technology used for signalling in the domain.

In our tests, the DT-IMS domain was the only one that needed to develop a CM, as the other domains were integrating it in the MS. The IMS Client manager (IMSCM) integrates all the functionality of a generic CM, but also controls the life-cycle (instantiation, destruction) of the IMS clients.

This IMSCM was contributed to the project as initial asset of DT and adapted to work in the framework developed.

The communication is established via WebSocket with JSON notation to allow web applications to get an own instance of an IMS user agent that runs in the cloud. The JSON protocol between the frontend application (M. Stub) and the IMSCM is simple and was easily extended to fit the needs of the project and its designed to avoid the need for any SIP Library in the browser.

3.4.3 Messaging Stub

The Messaging Stub (M. Stub) connects the client and the MS or CM via websockets technology and JSON notation. There is a different M. Stub for each one of the different servers. In some cases it can perform advanced features that are not supported by the MS/CM itself, as in the case of the Web-based DT MS, where the M. Stub is in charge of supporting multicast.

3.4.4 Identity Provider

The Identity Provider (IDP) is a mechanism used to retrieve the data from the users of a certain domain. Every domain will have their own IDP so they can easily maintain and update the data from their users. The IDP uses a REST API and the data is passed in JSON notation.

3.4.5 Library downloading mechanisms

In order to provide scalability and support for a large number of domains, the M. Stub will be downloaded during the execution of the application when it is needed. In order to do so, the identities provided by the IDP will have a field with the URL of the library to download, and once the library is downloaded, it will be shared for all the users of the domain related to the M. Stub to avoid unnecessary downloads.

3.4.6 Messaging format

One of the main interoperability problems is the different format of the messages and message flows adopted by the different domains, as WebRTC doesn't require an specific signalling protocol such as SIP. In order to address this problem, a new messaging format and flow was designed to fulfill the needs of the clients.

The flow and format of the messages will be translated by the M. Stub or MS into the interoperability framework compliant format.

The messages will follow JSON notation and the format shown in the figure 3.2.

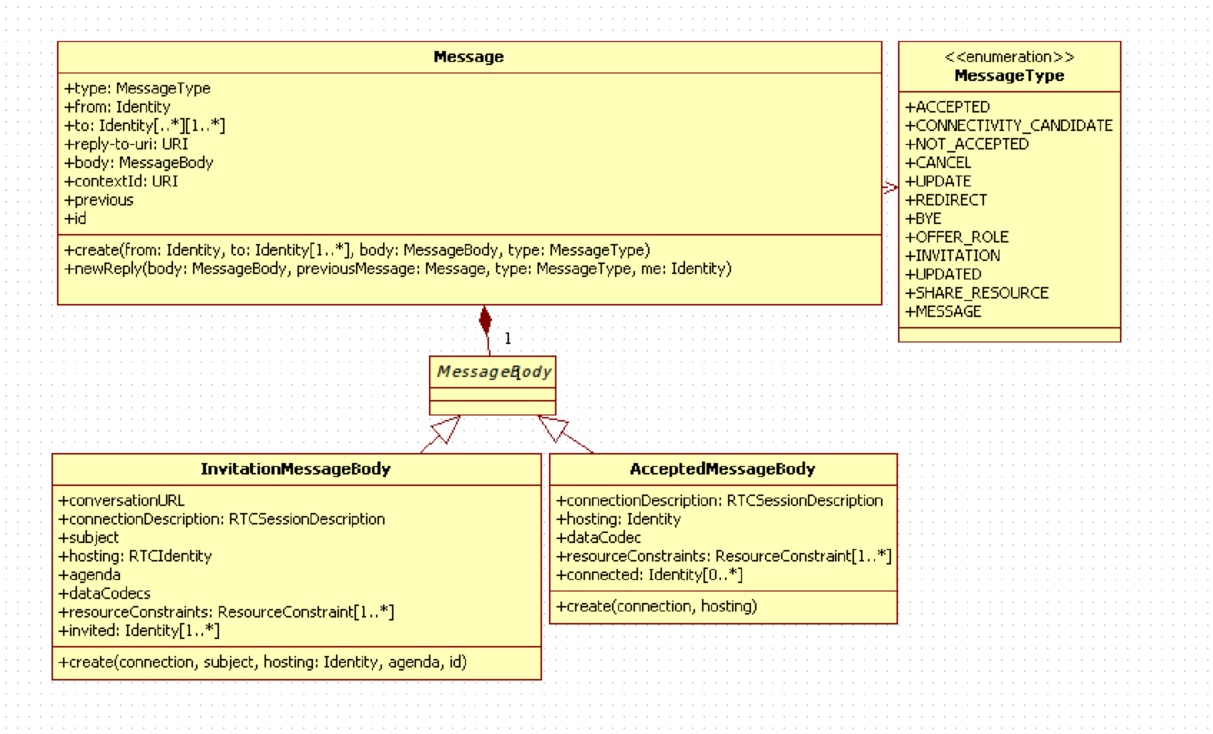

Message
+type: MessageType
+from: Identity
+to: Identity[..*][1..*]
+reply-to-uri: URI
+body: MessageBody
+contextId: URI
+previous
+id
+create(from: Identity, to: Identity[1..*], body: MessageBody, type: MessageType)
+newReply(body: MessageBody, previousMessage: Message, type: MessageType, me: Identity)
<<enumeration>>
MessageType
+ACCEPTED
+CONNECTIVITY_CANDIDATE
+NOT_ACCEPTED
+CANCEL
+UPDATE
+REDIRECT
+BYE
+OFFER_ROLE
+INVITATION
+UPDATED
+SHARE_RESOURCE
+MESSAGE
1
MessageBody
InvitationMessageBody
+conversationURL
+connectionDescription: RTCSessionDescription
+subject
+hosting: RTCIdentity
+agenda
+dataCodecs
+resourceConstraints: ResourceConstraint[1..*]
+invited: Identity[1..*]
+create(connection, subject, hosting: Identity, agenda, id)
AcceptedMessageBody
+connectionDescription: RTCSessionDescription
+hosting: Identity
+dataCodec
+resourceConstraints: ResourceConstraint[1..*]
+connected: Identity[0..*]
+create(connection, hosting)

Figure 3.2: Message class

3.4.7 Media proxies

The RTCWeb standard has defined two Mandatory to implement (MTI) audio codecs: opus and g.711. About the MTI video codecs, they are still not standardized. Google, Mozilla and Opera Software have positioned on the side of VP8 while Ericsson and Microsoft opt for H.264.

For this experimentation, only Google Chrome (so VP8 codec) was used, but in order to provide media compatibility with non-WebRTC SIP applications, a media proxy is needed. The Doubango Telepresence system is a conference system that was integrated on the IMS domain as an Application Server (AS) and integrates a media encoder that made possible to have a conference bridge with legacy SIP clients together with WebRTC clients using the interoperability framework developed.

3.4.8 Trickling support

What is trickling?

Internet Connectivity Establishment (ICE) Trickling is a protocol extension that allows ICE agents to send and receive candidates incrementally rather than exchanging complete lists. With such incremental provisioning, ICE agents can begin connectivity checks while they are still gathering candidates and considerably shorten the time necessary for ICE processing to complete. These connectivity candidates are descriptors of each connectivity option that can be potentially used by the other peer to connect.

Trickling support implementation

WebRTC uses the ICE Trickling mechanism by default, but some domains may not support it (e.g. IMS) To ensure the compatibility with clients that don't support trickling, the full SDP with all the candidates included is included in the last ICE candidate message that is otherwise empty. This way, the MS can obtain the full SDP and use it in SIP to ensure compatibility.

Chapter 4

System Architecture and Topologies

This chapter describes the interconnection between the different elements of the framework. The first section introduces the functional architecture for a generic case, to clarify the role of each server and its interconnections. The last section describes the different network topologies for each one of the use cases, which differ on the way the peers are connected to each other and to the servers.

The topologies explained in this chapter will be of special importance to understand the results of the tests performed, as the different topologies will have diverse results on interoperability and performance.

4.1 Functional Architecture

The functional architecture of the Web Real Time Communication (WebRTC) interoperability framework developed in this thesis is pictured in the Figure 4.1.

The main elements in the framework are the following:

Web App Server.
This web server stores and serves the frontend website to the peers, which will be executed locally in the peers' browser.

Identity Provider (IdP) Server.
This server contains and retrieves information about the Identities of the peers such as the name, the download address for the Messaging Stub (M. Stub) needed to connect with the peer, etc. All the queries to this server are made by the different frontend applications that are executed in the browser of the different peers.

Messaging Server (MS) / Client Manager (CM)
These servers route the signalling and can also make some additional tasks as call blocking,

accounting, etc. The main difference between a Messaging Server (MS) and a Client Manager (CM) is that the first receives connections from peers from other domains to contact peers of its domain and the latter receives connections from peers of the same domain to receive calls from any domain or to make calls to other peers in the same domain. In most domains, Client Manager's function is integrated into the Messaging Server.

Media GW

Also known as Multipoint Control Unit (MCU), this server mixes and transcodes the video/audio streams. It is only used in the Media Stream Star Topology (see 4.2.2) or to achieve media format compatibility between WebRTC clients and legacy IP Multimedia Subsystem (IMS) clients.

As shown in the Figure 4.1, all the multimedia communication is transmitted directly between peers (except for the case of the Media GW). The traffic with the rest of the servers is only signalling or small data, so they can support a large amount of clients. Its advised to avoid, as far as possible, the use of Media GW as they need large amounts of bandwidth and computational power to serve even a small/medium number of clients.

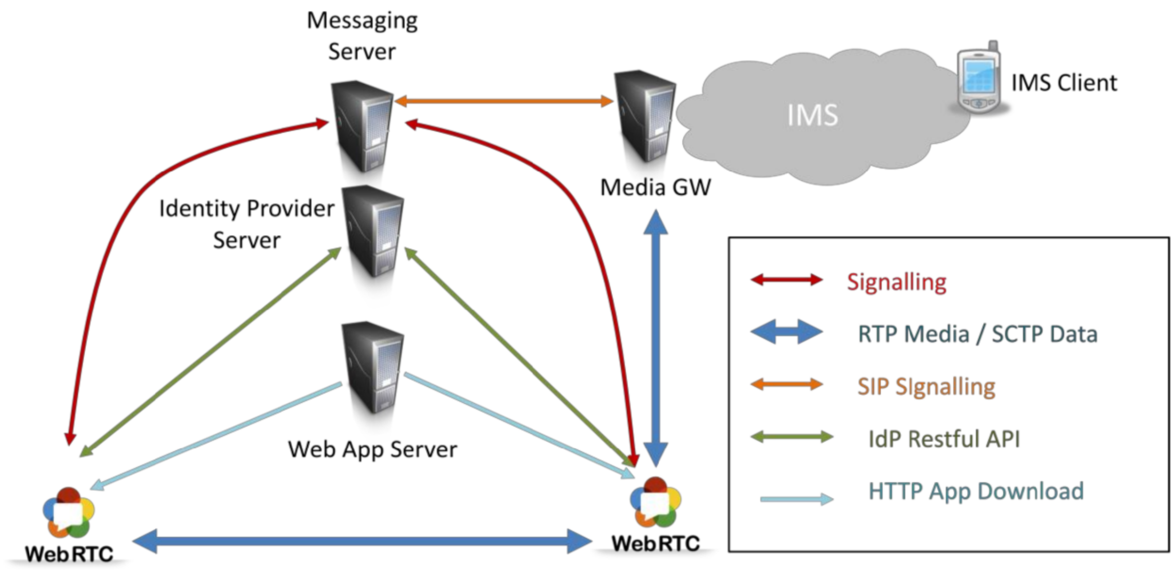


Figure 4.1: Functional architecture

4.2 Network Topologies

To fully understand the concepts described in this section, we have to keep in mind that during the initialization of the client it will connect to the Client Manager (CM) (integrated in the

Messaging Server (MS) in most of the cases) of its domain. This will connect the client with its domain enabling it to receive (or send) signalling messages.

4.2.1 2 party conversations

For 2 party conversations, the following 3 models were analyzed:

1. Caller connects to callee MS (Figure 4.2).

   This model is the one adopted in the project. The caller will always connect to the callee's MS (or CM for intra-domain calls). In order to do so, the application will download the library for its M. Stub.

   The main advantage of this system is that for each peer you only use one M. Stub/MS, which is the same for caller and callee, which minimizes complexity and traffic. This also minimizes the consequences in case of a server failure or overload, having impact on calls to that domain but no effect on the calls clients make to other domains. Another advantage is that all the logic belongs to the client part not overloading the MS, if the MS is updated only the library of the M. Stub has to be updated, minimizing maintenance costs.

   The disadvantages of this model are loss of control by the local domain to charge, account or restrict the calls made to other domains and the fact that the resources are spent in the domain of the callee and not on the caller side. Another important disadvantage is the impossibility of a legacy client from a domain to call to another domain, as legacy clients don't implement the M. Stub mechanism to connect to the foreign MS.

   This solution was chosen because the main requirements are to minimize network resources spent and simplicity to reduce implementation and maintenance costs.

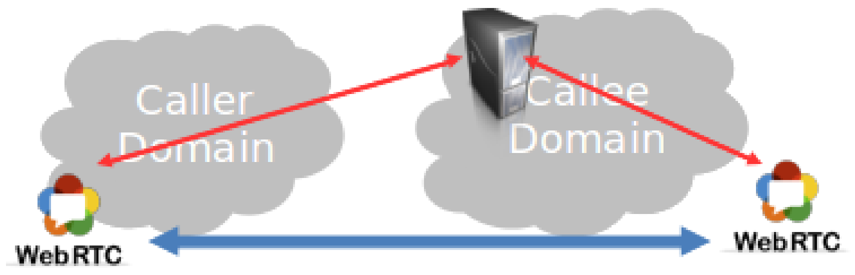


Figure 4.2: Proposed MS topologies: Caller connects to callee MS

2. Both peers connect to local CM. MS communication (Figure 4.3).

   The caller will always connect to the local domain and the local CM will forward the messages to the callee domain via remote MS. This implies the domains have to be able to interact with each other.

The advantages of this system are that the local domain still keeps track and control calls even if they are addressed to another domain; it also allows legacy clients to connect to foreign domain clients and removes the need of downloading any library for the M. Stub, as the client will only connect to the local domain CM.

The disadvantages of this model are that all the load falls on the servers that have to route and process the signaling, using resources from both caller and callee domains and increasing the number of failure points. It also difficult the maintenance and scalability, as a communication system between servers should be designed and changes in one of the domains can imply changes which have to be tested and approved by every domain.

This architecture was discarded because of the high load and network resources spent in the link between domains and also because its complexity and limited scalability increases maintenance costs.

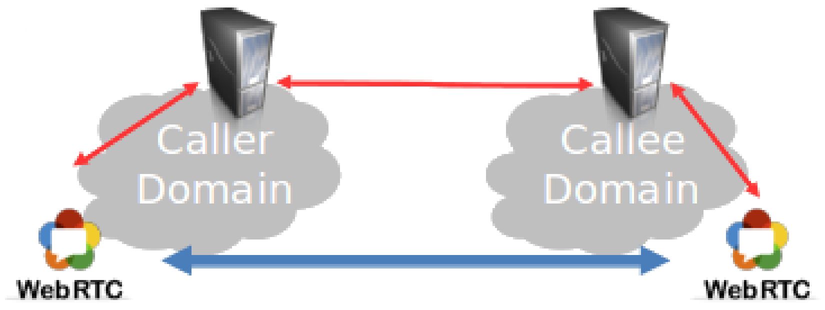


Figure 4.3: Proposed MS topologies: MS communication

3. Both peers connect to remote MS (Figure 4.4).

   Both caller and callee will always connect to the remote MS to send signalling messages which will be received via the local domain CM.

   This solution is identical to the one described on "Caller connects to callee MS (Figure 4.2)" but using a symmetrical topology. This makes the logic easier as it doesn't matter whether the peer is the caller or the callee, it will always send signaling messages to the remote MS and receive via the local CM. This is specially useful in multiparty conversations where the message flow gets more complicated.

   On the other hand, this topology uses both MS, increasing the failure points and generates traffic on N^2 links while the first one uses only $N^2-N-(1+2+3+...+(N-2)) = \frac{N^2}{2}+\frac{N}{2}-1$ links, as in the second case the link from CM to the peer is shared for all the connections established to the MS with that user as callee.

   For this reason this architecture was also discarded.

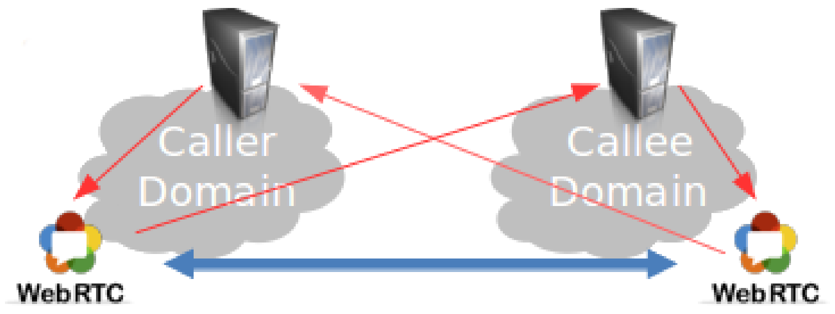


Figure 4.4: Proposed MS topologies: Both peers connect to remote MS

4.2.2 Multiparty conversations

When a conversation has more than 2 peers, the challenge is not only how to connect caller and callee's (as it was explained in the previous section), but also how all the callees will communicate to each other.

In this section 3 different mechanisms will be analyzed, the 2 first focusing on the approach for signaling (centralized or peer to peer) and the last one on centralized media/data connections.

Mesh multiparty conversation with hosting

In this configuration, the participant that starts the conversation will offer his MS for all the other peers to connect to it and be the hosting for all the signaling messages.

It was successfully implemented for intradomain, but not working for those cases where the host doesn't support the publish mechanism.

A simplification of this algorithm that avoided the need of using a publish mechanism (see Appendix B.2) was taken into consideration, but it was decided to implement this one in order to analyze the implementation of this mechanism either in the MS or in the M. Stub.

The description of the algorithm for multiparty conversations that was implemented in this thesis is explained in Appendix B.1.

No Hosting

The Figure 4.6 shows the multiparty topology with no hosting. In this topology there is no Hosting peer, all the peers connect to every other peer via their remote MS.

Even though the message format and API design support no hosting architecture, the development for multiparty conversations with hosting was prioritized, as it uses less network connections and it is generally more efficient. For this reason, all the Mesh multiparty tests explained in the next chapters, refer to the case where a participant hosts the conversation.

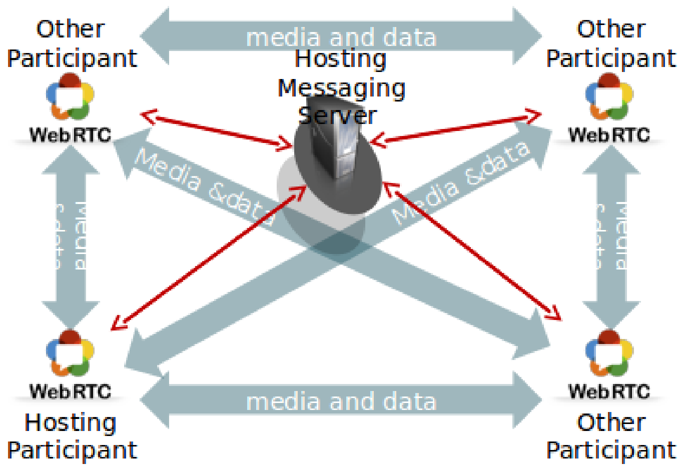


Figure 4.5: Mesh multiparty with hosting topology

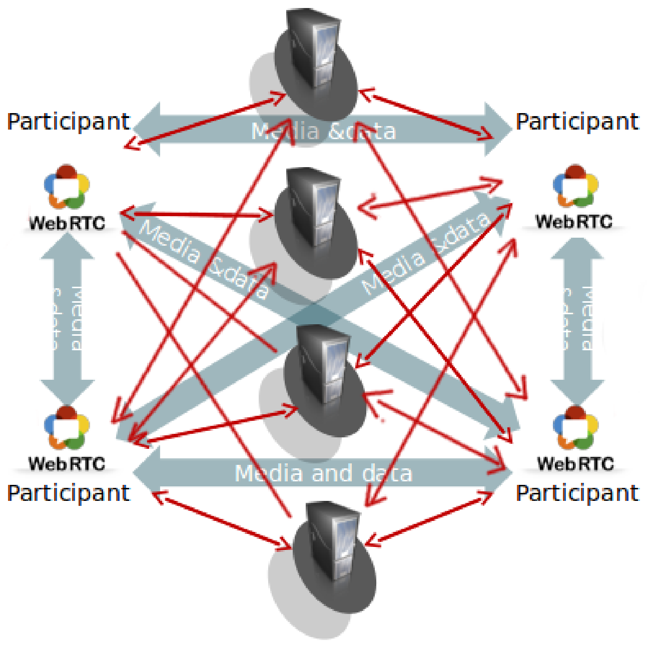


Figure 4.6: Multiparty with no hosting topology

Media Stream Star Topology

Media Stream Star is a topology where all the participants use servers from the same domain to communicate and a central media server for the media and data streams. This central media server receives the media and data streams from the peers and mixes and distributes the resulting stream among the peers, as pictured in Figure 4.7.

In order to implement this topology, the open source Doubango Telepresence [32] project was used. It was deployed as an Application Server (AS) in Deutsche Telekom (DT)-IMS domain, acting as a bridge for calls. All clients from the different domains should call this bridge, so DT-IMS servers will be used for signalling and the Telepresence AS will act as a central media server, mixing the media streams.

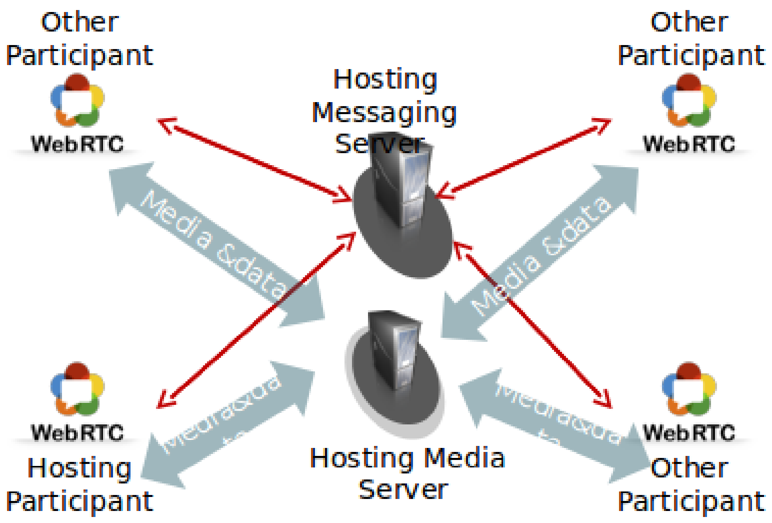


Figure 4.7: Media Stream Star topology

Chapter 5

Application API

In order to build interoperable Web Real Time Communication (WebRTC) applications, it became necessary to develop an API that held all the interoperability mechanisms together. This chapter explains the API developed for the interoperability framework where the two first sections emphasize on the entities that form it and its specification. This is important to understand how the API is divided in two different layers depending on the need of implementing low level WebRTC applications or introducing an abstraction layer to remove the complexity of WebRTC. Last two sections explain the development of WebRTC applications based on the API previously described to show its capacity to create interoperable applications from the point of view of the developers.

The analysis and description made in this chapter show the potential of this framework not only for large commercial products but also for small developments, adding value to its use and expanding the interoperable ecosystem for WebRTC.

5.1 Entities

The main entities in the interoperability framework API in the client are (Figure 5.1):

Identity. It represents an user and contains all information needed to support Conversation services including the service endpoint to retrieve the protocol stack (Messaging Stub) that will be used to establish a signalling channel with the Identity domain Messaging Server (MS). The Identity entity extends the current Identity concept defined in WebRTC specification [15] to support seamless interoperability by using the Signalling on-the-fly mechanism.

MessagingStub. It implements the protocol Stack used to communicate with a certain Messaging Server (MS).

Message. It is used to exchange all data needed to setup, update and close media and data connection between peers via the Messaging Server (MS). It may also be used for other purposes e.g. presence information management.

Conversation. Class that manages all participants including the setup, update or close of media and data connections.

Participant. Class that handles all operations needed to manage the participation of an Identity (User) in a conversation including the WebRTC PeerConnection functionalities. The Local Participant is associated with the Identity that is using the Browser while the Remote Participant is associated to remote Identities (users) involved in the conversation.

Resource. Class that represents the digital assets that are shared among participants in the conversation including participants' voice, video, screens, photos, video Clips, music clips, documents, etc. These assets are usually managed by the Participant that owns it. For local participants assets are sent (e.g. WebRTC outgoing stream tracks) while for remote participants assets are received (e.g. WebRTC incoming stream tracks). Some Resource types like Chat are not managed by a Participant but by the Conversation.

Data Codec. It is used by Resources that are shared on top of the Data Channel, like file sharing and Textual Chat, to decode and encode the data in a consistent way by all the peers. The Data Codec may also be downloaded on-the-fly by the peers.

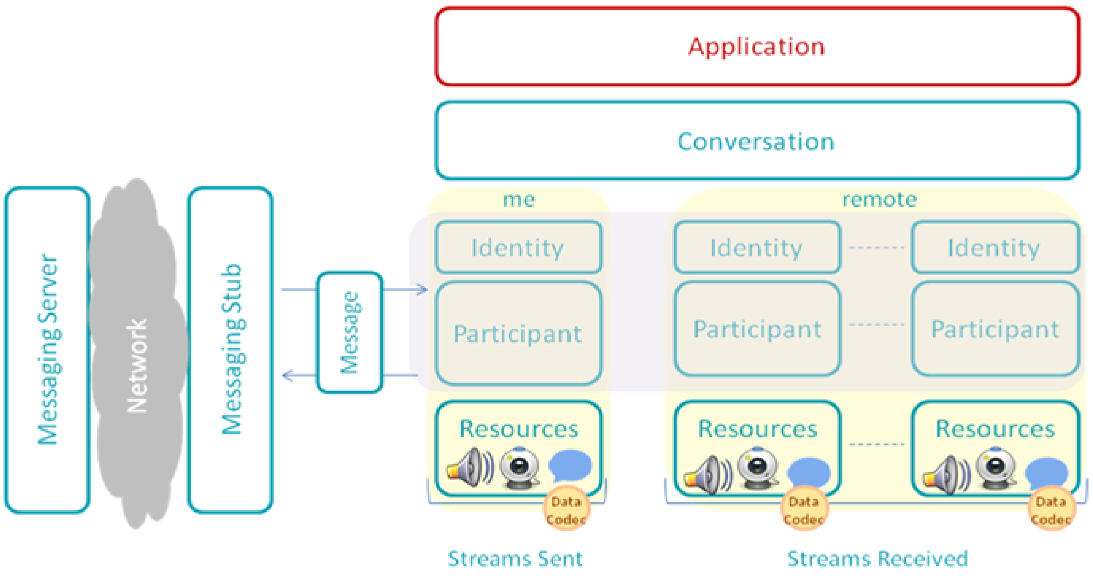


Figure 5.1: Main API Classes

5.2 Specification

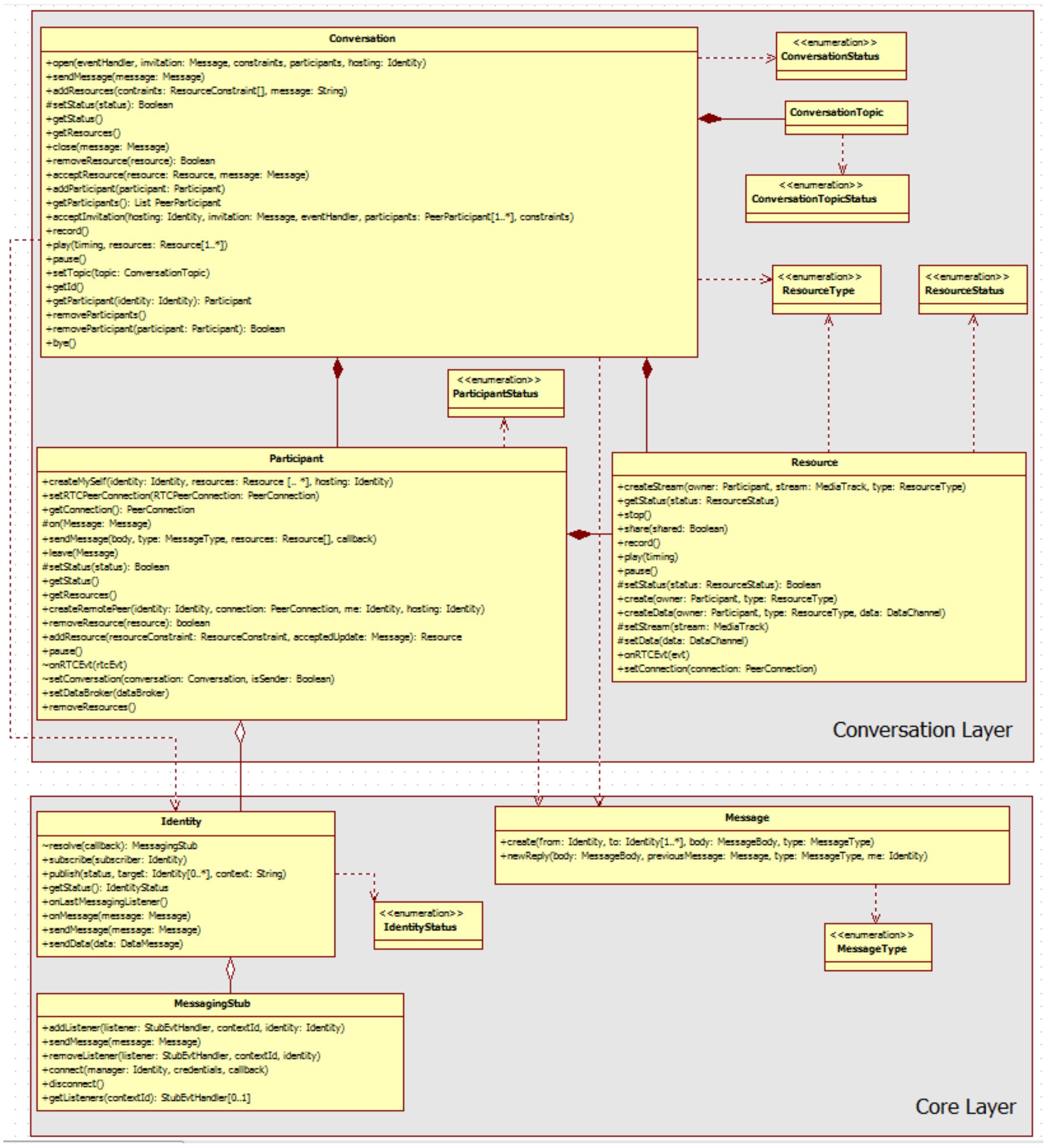

Conversation
+open(eventHandler, invitation: Message, constraints, participants, hosting: Identity)
+sendMessage(message: Message)
+addResources(contraints: ResourceConstraint[], message: String)
#setStatus(status): Boolean
+getStatus()
+getResources()
+close(message: Message)
+removeResource(resource): Boolean
+acceptResource(resource: Resource, message: Message)
+addParticipant(participant: Participant)
+getParticipants(): List PeerParticipant
+acceptInvitation(hosting: Identity, invitation: Message, eventHandler, participants: PeerParticipant[1..*], constraints)
+record()
+play(timing, resources: Resource[1..*])
+pause()
+setTopic(topic: ConversationTopic)
+getId()
+getParticipant(identity: Identity): Participant
+removeParticipants()
+removeParticipant(participant: Participant): Boolean
+bye()
<<enumeration>> ConversationStatus
ConversationTopic
<<enumeration>> ConversationTopicStatus
<<enumeration>> ResourceType
<<enumeration>> ResourceStatus
<<enumeration>> ParticipantStatus
Participant
+createMySelf(identity: Identity, resources: Resource [.. *], hosting: Identity)
+setRTCPeerConnection(RTCPeerConnection: PeerConnection)
+getConnection(): PeerConnection
#on(Message: Message)
+sendMessage(body, type: MessageType, resources: Resource[], callback)
+leave(Message)
#setStatus(status): Boolean
+getStatus()
+getResources()
+createRemotePeer(identity: Identity, connection: PeerConnection, me: Identity, hosting: Identity)
+removeResource(resource): boolean
+addResource(resourceConstraint: ResourceConstraint, acceptedUpdate: Message): Resource
+pause()
~onRTCEvt(rtcEvt)
~setConversation(conversation: Conversation, isSender: Boolean)
+setDataBroker(dataBroker)
+removeResources()
Resource
+createStream(owner: Participant, stream: MediaTrack, type: ResourceType)
+getStatus(status: ResourceStatus)
+stop()
+share(shared: Boolean)
+record()
+play(timing)
+pause()
#setStatus(status: ResourceStatus): Boolean
+create(owner: Participant, type: ResourceType)
+createData(owner: Participant, type: ResourceType, data: DataChannel)
#setStream(stream: MediaTrack)
#setData(data: DataChannel)
+onRTCEvt(evt)
+setConnection(connection: PeerConnection)
Conversation Layer
Identity
~resolve(callback): MessagingStub
+subscribe(subscriber: Identity)
+publish(status, target: Identity[0..*], context: String)
+getStatus(): IdentityStatus
+onLastMessagingListener()
+onMessage(message: Message)
+sendMessage(message: Message)
+sendData(data: DataMessage)
<<enumeration>> IdentityStatus
Message
+create(from: Identity, to: Identity[1..*], body: MessageBody, type: MessageType)
+newReply(body: MessageBody, previousMessage: Message, type: MessageType, me: Identity)
<<enumeration>> MessageType
MessagingStub
+addListener(listener: StubEvtHandler, contextId, identity: Identity)
+sendMessage(message: Message)
+removeListener(listener: StubEvtHandler, contextId, identity)
+connect(manager: Identity, credentials, callback)
+disconnect()
+getListeners(contextId): StubEvtHandler[0..1]
Core Layer

Figure 5.2: Main API Classes overview and dependencies

The Figure 5.2 shows a general overview of the main API classes and their dependencies. Logically, the API can be separated into two main layers – the basic Core layer on the bottom and an Conversation layer on top of it. These layers and the API classes belonging to them are highlighted in the figure.

5.2.1 Core Layer

This layer includes all classes implementing the basic interoperability concepts as described in the Section 5.1 Entities. This includes especially the mechanisms for on-the-fly loading of signalling libraries to exchange messages between different domains.

Identity class

The Identity represents a user and contains all information needed to support Conversation services including the service endpoint to retrieve the protocol stack (Messaging Stub (M. Stub)) that will be used to establish a signalling channel with the Identity domain MS.

Note that Identities are only created by using the corresponding create-methods of the Identity Provider (IDP).

Methods

resolve(callback)

This method downloads a M. Stub and keeps a reference to it in a local attribute, if not already done before. That means the download will only be performed once. After download it invokes the given callback with a reference to the downloaded M. Stub.

Parameters:

Name	**Type**	**Description**
callback	callback(MessagingStub)	callback that is invoked with M. Stub as param; if download failed then the M. Stub param is empty

IdP class

The IDP is a basic implementation of an identity provider. Its main purpose is to create and maintain Identities and their relation to the corresponding M. Stub. The IDP is a singleton object, i.e. there is always just one instance of it.

Methods

<static> getInstance(rtcIdentity, options)

This is a getter for an already created instance of the IdP. The params are optional. In case there was no instance already created before, the params can also be given here and will then be used for initial creation of the object.

Parameters:

Name	**Type**	**Description**
rtcIdentity	URI	callback that is invoked with M. Stub as param; if download failed then the M. Stub param is empty
options	Object	options that influence the behavior of the Idp

createIdentities(rtcIdentities, onSuccessCallback)

This method takes either a single rtcIdentity or an array of rtcIdentities and creates Identity objects from them. The successfully created Identities are then returned in an Array in the success callback. Note that if one or more rtcIdentities can't be created then the returned array is shorter than the given array.

Parameters:

Name	**Type**	**Description**
rtcIdentities	Array	list of rtcIdentities for which Identities shall be created
onSuccessCallback	callback(Array)	the callback that is invoked with the resulting array of Identities (can be shorter than the input list)

createIdentity(rtcIdentity, onSuccessCallback, onErrorCallback)

This method takes a single rtcIdentity and creates an Identity from it. The successfully created Identity is then returned as parameter of the success callback.

Parameters:

Name	**Type**	**Description**
rtcIdentity	URI	the rtcIdentity for which an Identity shall be created
onSuccessCallback	callback(Identity)	the callback that is invoked with the resulting Identity
onErrorCallback	callback()	a callback that is invoked in case of any errors during this process

MessagingStub

The M. Stub implements the protocol Stack used to communicate with a certain MS. It defines a set of methods that must be implemented in order to support a new domain.

Methods

connect(ownRtcIdentity, credentials, callbackFunction)

Creates the connection, connects to the server and establish the callback to the listeners on new message.

Parameters:

Name	**Type**	**Description**
ownRtcIdentity	URI	URI with the own RTCIdentity used to connect to the MS.
credentials	Object	Credentials to connect to the server.
callbackFunction	callback()	Callback to execute when the connection is done.

disconnect()

Disconnects from the server.

Parameters:

Name	**Type**	**Description**
rtcIdentities	Array	list of rtcIdentities for which Identities shall be created
onSuccessCallback	callback(Array)	the callback that is invoked with the resulting array of Identities (can be shorter than the input list)

sendMessage(message)

Sends the specified message.

Parameters:

Name	**Type**	**Description**
message	Message	Message to send.

Message

This class is a data-holder for all messages that are sent between the domains.

Constructor

new Message(from, to, body, type, context)

Parameters:

Name	**Type**	**Description**
from	Identity	Sender of the message
to	Identity[]	Recipients of the message
body	MessageBody	Message body (a json struct)
type	MessageType	Type of the Message (@see MessageType)
context	string	ID of the conversation. (Optional. For conversation related messages it is mandatory.)

Message Type

Enumeration for the Message Types

Name	**Type**	**Default**	**Description**
INVITATION	string	invitation	Message to invite a peer to a conversation.
ACCEPTED	string	accepted	Answer for conversation accepted.
CONNECTIVITY_CANDIDATE	string	connectivityCandidate	Message contains connectivity candidate.
NOT_ACCEPTED	string	notAccepted	Answer for conversation not accepted.
CANCEL	string	cancel	Message to cancel an invitation.
ADD_RESOURCE	string	addResource	Message to add a Resource to the conversation.
REMOVE_PARTICIPANT	string	removeParticipant	Message to remove a Participant from the conversation.
BYE	string	bye	Message to finish the communication with a peer.
UPDATE	string	update	Message to add a new Resource.
UPDATED	string	updated	Answer to add a new Resource.

MessageFactory

This class creates messages which are compliant with the interoperability protocol. Please note that all functions in this class are static, so there is no need to create MessageFactory objects.

Methods

<static> Message createAnswerMessage(from, to, contextId, constraints, hosting, connected)

Creates an Answer message, the connectionDescription field will be empty and has to be filled before sending.

Parameters:

Name	Type	Description
from	Identity	The Identity that figures as sender of the message.
to	Identity[]	The Array of Identity that figures as receiver of the message.
contextId	string	The contextId of the conversation related to the invitation.
constraints	ResourceConstraints	The resource constraints for the resources initialized on conversation start.
hosting	Identity	The host of the conversation (optional). [NOT IMPLEMENTED, by default the host will be the one starting the conversation
connected	Identity[]	Array of Identity that are already connected to the conversation. Used to establish the order in the connection flow for multiparty.

Returns: The created Message

<static> Message createCandidateMessage(from, to, contextId, label, id, candidate, lastCandidate)

Creates a Message containing an Internet Connectivity Establishment (ICE) candidate

Parameters:

Name	**Type**	**Description**
from	Identity	The Identity that figures as sender of the message.
to	Identity[]	The Array of Identity that figures as receiver of the message.
contextId	string	The contextId of the conversation related to the invitation.
label	string	The label of the candidate.
id	string	The id of the candidate.
candidate	string	The ICE candidate string.
lastCandidate	boolean	Boolean indicating if the candidate is the last one. If true, include the full SDP in the candidate parameter for compatibility with domains that don't support trickling.

Returns: The created Message

<static> createInvitationMessage(from, to, contextId, constraints, conversationURL, subject, hosting)

Creates an Invitation message, the connectionDescription field will be empty and has to be filled before sending.

Parameters:

Name	**Type**	**Description**
from	Identity	The Identity that figures as sender of the message.
to	Identity[]	The Array of Identity that figures as receiver of the message.
contextId	string	The contextId of the conversation related to the invitation.
constraints	ResourceConstraints	The resource constraints for the resources initialized on conversation start.
conversationURL	string	The URL of the conversation (optional).
subject	string	The subject of the conversation. (optional).
hosting	Identity	The host of the conversation (optional).

Returns: The created Message

<static> createUpdateMessage(from, to, contextId, newConstraints)

Creates an Update message, the newConnectionDescription field will be empty and has to be filled before sending.

Parameters:

Name	**Type**	**Description**
from	Identity	The Identity that figures as sender of the message.
to	Identity[]	The Array of Identity that figures as receiver of the message.
contextId	string	The contextId of the conversation related to the invitation.
newConstraints	ResourceConstraints	The resource constraints for the resources to update.

Returns: The created Message

<static> createUpdatedMessage(from, to, contextId, newConstraints)

Creates an Updated message, the newConnectionDescription field will be empty and has to be filled before sending.

Parameters:

Name	**Type**	**Description**
from	Identity	The Identity that figures as sender of the message.
to	Identity[]	The Array of Identity that figures as receiver of the message.
contextId	string	The contextId of the conversation related to the invitation.
newConstraints	ResourceConstraints	The resource constraints for the resources to update.

Returns: The created Message

5.2.2 Conversation Layer

The application support layer provides developers with a high-level API that encapsulates a lot of the complex functionalities required for WebRTC communication apps. This includes a full encapsulation of the complex WebRTC and RTCPeerConnection specific coding as well as for the establishment of multi-party communication sessions.

The goal of this support layer is to allow very fast implementation of communication apps without deeper knowledge of the complex underlying technology.

Conversation

Class that represents a conversation between 2 or more peers.

Constructor

new Conversation(participants, id, owner, hosting, rtcEvtHandler, msgHandler)

Parameters:

Name	**Type**	**Description**
participants	Participant[]	list of Participant involved in the conversation.
id	string	Unique Conversation identification.
owner	Participant	the Participant organizing the conversation.
hosting	Identity	the Identity that is providing the signalling message server.
rtcEvtHandler	callback(rtcEvent)	Event handler implemented by the Application to receive and process RTC events triggered by WebRTC Media Engine
msgHandler	callback(Message)	Message handler implemented by the Application to receive and process Messages from the M. Stub

Methods

addParticipant(participant, invitation)

Adds a participant to the conversation.

Parameters:

Name	**Type**	**Description**
participant	Participant	the Participant to add to the conversation.
invitation	string	the invitation to be attached to the Message body.

addResource()

Add a Resource to this Conversation including all the signaling and logical actions required.

bye()

The current user invokes bye, if he wants to leave an ongoing conversation. Other participants might stay in this conversation, in case that it was a multi-party call with more participants. Sends a REMOVE_PARTICIPANT message to ALL participants and sets the conversation status to CLOSED.

boolean close(message)

Close the conversation with the given message. Sends this message to ALL participants and sets the conversation status to CLOSED.

Parameters:

Name	**Type**	**Description**
message	Message	the final message to be sent to ALL participants of this conversation.

Returns: True if successful, false if the participant is not the owner. Type boolean

getStatus() Returns the status of this conversation.

open(rtcIdentity, invitation) A Conversation is opened for invited participants. Creates the remote participant, resolves and gets the stub, creates the peer connection, connects to the M. Stub and sends invitation.

Parameters:

Name	**Type**	**Description**
rtcIdentity	string[]	list of users to be invited
invitation	string	body to be attached to INVITATION MESSAGE

sendMessage(message) If to-field of the message is empty, then send message to all participants, send only to specified participants if to-field is filled. (Message.to-field is a list of identities.)

Parameters:

Name	**Type**	**Description**
message	Message	the Message to be sent to the specified Identities or or ALL participants.

Participant

The Participant class handles all operations needed to manage the participation of an Identity (User) in a conversation including the WebRTC PeerConnection functionalities. The Local Participant is associated with the Identity that is using the Browser while the Remote Participant is associated to remote Identities (users) involved in the conversation.

Methods

addResource(resourceConstraints, message, callback, errorCallback)

Adds a Resource to this participant including all the signaling and logical actions required.

Parameters:

Name	**Type**	**Description**
resourceConstraints	resourceConstraints[]	Array of constraints for the initial resources of the remote participant (CURRENT IMPLEMENTATION WILL TAKE THE FIRST ONE).
message	Message	In case an UPDATE message is received, it should be passed to this function as a parameter to process it and send the UPDATED.
callback	callback	Callback function fired when the resource was added succesfully.
errorCallback	errorCallback	Callback function fired when an error happens.

createRemotePeer(identity, myParticipant, contextId, resourceConstraints, rtcEvtHandler, msgHandler, iceServers)

Creates a remote participant.

Parameters:

Name	**Type**	**Description**
identity	Identity	Identity of the participant
myParticipant	Participant	Participant representing the local user of the application.
contextId	string	Identifier of the conversation this participant belongs to.
resourceConstraints	resourceConstraints[]	Array of constraints for the initial resources of the remote participant. (CURRENT IMPLEMENTATION WILL TAKE THE FIRST ONE)
rtcEvtHandler	onRTCEvt	Callback function that handles WebRTC Events.
msgHandler	onMessage	Callback function that handles signaling Events.
iceServers	RTCIceServer	Configuration parameters for ICE servers http://www.w3.org/TR/webrtc/#widl-RTCConfiguration-iceServers

getResources(resourceConstraints, resourceType, id)

Searches and retrieves Resources.

Parameters:

Name	**Type**	**Description**
resourceConstraints	resourceConstraints	Searches the resources by constraints (OPTIONAL)
resourceType	resourceType	Searches the Resources by type. (OPTIONAL)
id	string	Searches the Resources by ID.

RTCPeerConnection getRTCPeerConnection()

Returns a reference to the RTCPeerConnection that is established with this participant.

Returns: PeerConnection, the connection attribute for a participant

ParticipantStatus getStatus() Returns the current status of this Participant

Returns: ParticipantStatus ... gets the status attribute for a participant

leave(sendMessage)

The Participant leaves the Conversation removing all resources shared in the conversation. Participant status is changed accordingly.

Parameters:

Name	Type	Description
sendMessage	boolean	If true a BYE message will be sent to the participant before removing it. If false the participant will be removed locally from the conversation without sending any message

leave(messageBody, messageType, constraints, callback, errorCallback)
The method will create the message and send it to the participant.
Parameters:

Name	Type	Description
messageBody	MessageBody	The body of the message (depends on the MessageType)
messageType	MessageType	The type of the message.
constraints	ResourceConstraints	For the messages that imply information about the Resources, constraints about them (OPTIONAL)
callback	callback	Callback for successful sending.
errorCallback	errorCallback	Error Callback.

setDataBroker(databroker)
SetDataBroker - Sets the @DataBroker to a Participant
Parameters:

Name	Type	Description
databroker	DataBroker	DataBroker to set.

Resource

The Resource class represents the digital assets that are shared among participants in the conversation.

Constructor
new Resource(resourceConstraint, codec)
Parameters:

Name	**Type**	**Description**
resourceConstraint	ResourceConstraint	Constraints of the Resource. Object with the following syntax {type: ResourceType, constraints: codec or MediaStreamConstraints}.
codec	Codec	For data types only, Codec used.

DataBroker

The DataBroker Class handles all the operations to choose the right codecs and the channels to send, receive and handle messages related to the Data Channel.

Constructor
new DataBroker()

Methods:
addCodec(codec) Adds a Codec to the DataBroker.
Parameters:

Name	**Type**	**Description**
codec	Codec	Codec used.

addDataChannel(dataChannel, identity) Adds a DataChannel to the DataBroker with the respective identity
Parameters:

Name	**Type**	**Description**
codec	DataChannel	A DataChannel object.
identity	Identity	An Identity object.

onDataChannelEvt(dataMessage) Receives a Message from a Data Channel and It will forward received data to the appropriate Codec based on codecId set in the DataMessage.
Parameters:

Name	**Type**	**Description**
dataMessage	DataMessage	JsonObject with the content of a message, is a DataMessage Type.

removeCodec(codec) Removes a Codec from the DataBroker.
Parameters:

Name	**Type**	**Description**
codec	Codec	Codec to remove.

removeDataChannel(identity) Removes a DataChannel from the DataBroker
Parameters:

Name	**Type**	**Description**
identity	Identity	An Identity object.

removeDataChannel(identity) Removes a DataChannel from the DataBroker
Parameters:

Name	**Type**	**Description**
identity	Identity	An Identity object.

send() Sends a Message from channel in the DataBroker.
Parameters:

Name	**Type**	**Description**
string	string	String with the content of a message

Codec

The Codec Class handles all the operations needed to manage a codec used in a conversation. This codec is used for Data connections with WebRTC Datachannels, used for chat, filesharing, etc.

Constructor
new Codec()

Methods:
addListener(listener) Adds a Listener to the codec that will handle all the messages to the application.
Parameters:

Name	Type	Description
listener	Listener	A Listener function that should be implemented in the application.

getReport() getReport function.
onData(dataMessage) Listener to receive DataMessages from data broker.
Parameters:

Name	Type	Description
dataMessage	DataMessage	JsonObject with the content of a message, is a DataMessage Type.

addDataChannel(dataChannel, identity) Adds a DataChannel to the DataBroker with the respective identity
Parameters:

Name	Type	Description
codec	DataChannel	A DataChannel object.
identity	Identity	An Identity object.

removeListener(listener) Removes a Listener to the codec that will handle all the messages to the application.
Parameters:

Name	Type	Description
listener	Listener	The listener that should be removed.

saveToDisk(fileUrl, fileName) Store the received files in the harddisk.
Parameters:

Name	Type	Description
fileUrl	FileUrl	File URL where the applicationr receives the information about the file.
fileName	string	String which should contain the name to the file to store in the disk.

send(data) Operation to send data e.g. a Chat message, File.
Parameters:

Name	Type	Description
data	string	String with the content of a message, is a simple string even for chat and file.

setDataBroker(dataBroker) Adds a DataBroker to the codec.
Parameters:

Name	Type	Description
dataBroker	DataBroker	A DataBroker object.

5.3 Development of applications based on the API

This section illustrates the main characteristics of the API from the point of view of the developer and shows how to create WebRTC applications which are compatible with the interoperability framework. The API introduced in this chapter provides different entry points which differ in the level of abstraction and complexity. These levels are illustrated in Figure 5.3.

The **Conversation layer** provides the highest level of abstraction and hides all the complexity of programming a WebRTC application. This includes methods for accessing media sources, establishment and management of RTCPeerConnections, abstraction of the call-participants and the whole signalling between them. Therefore this option is most suitable for developers who want to start an application from scratch and in the most simple and straight-forward way. Nevertheless this option provides full control of all parameters and flexible ways for modification

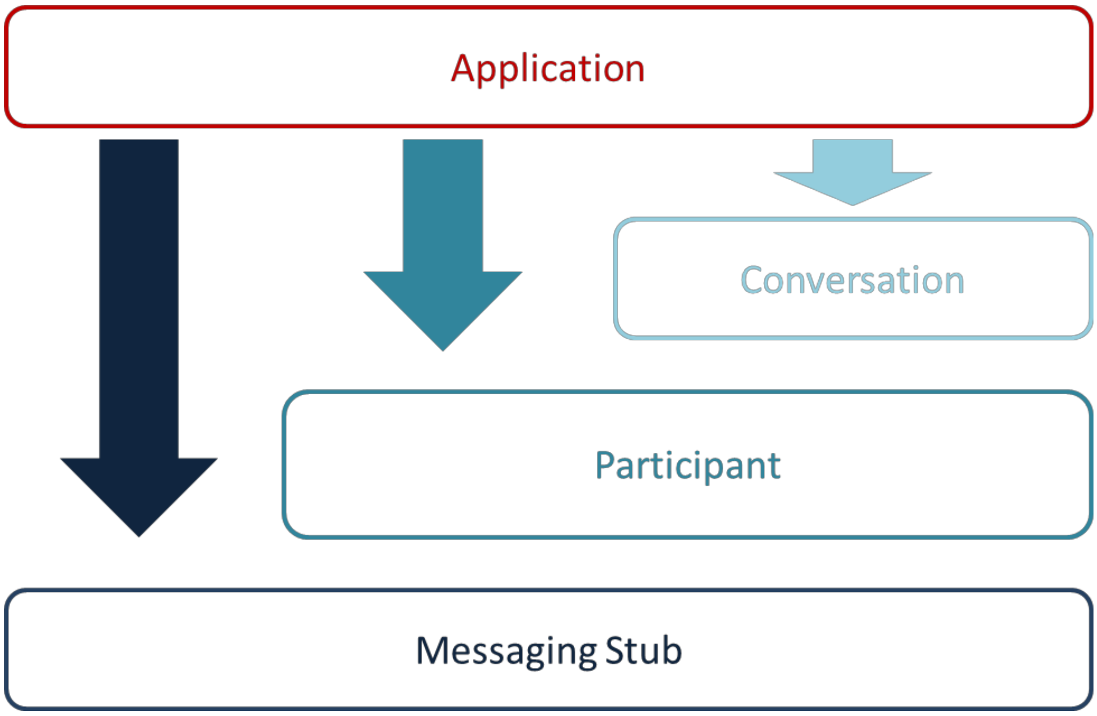


Figure 5.3: Different API abstraction layers.

of running conversations. A code example of a simple application using this layer of the API can be found on the Appendix C.1.

The other extreme of programming is to use the **Core layer** directly. This method provides mechanisms for handling identities and for the exchange of standardized messages for the establishment of WebRTC communications. This also includes the described on-the-fly methods for downloading of Messaging Stubs and therefore provides the advantage of cross-domain interoperability. However all WebRTC related coding and the management of calls and their participants are left to the programmer. This option is intended for developers who already have a WebRTC application and want to make use of the interoperability features. An example of a simple application coded using the Core layer is shown on the Appendix C.2

There is also a third option – the **Participant layer** – which is a compromise between both options described above. It provides an abstraction of the participants of a conversation and handles all WebRTC related stuff for them, but it does not provide an abstraction of a conversation itself. So it might be of interest for developers who don't want to struggle with the complex WebRTC coding, but want to keep their own concept of what a Conversation is.

5.4 Messaging Stub Development

A Messaging Stub (M. Stub) for a certain domain acts as the "glue" between the API messages and the message-format and signalling protocol of a certain domain. It implements the domain specific part of the signalling. The overall concept is that applications just deal with the messages specified by the framework and don't need to care about the details of signalling. The main translation job shall be done by the implementation of the M. Stub. There are 3 methods that a developer has to implement.

Connect function

With the connect method, the M. Stub implements the special means to connect to the MS of the corresponding application domain. This depends on the signalling protocol for the domain, which is a domain internal agreement between the MS and M. Stub. The first parameter is an rtcIdentity, which is a URI indicating the identity of the User that is currently running the application in the browser. The code attached in the appendix C.3.1 shows an example of a connect() method for a simple WebSocket based domain. This example does not perform any special authentication/authorization mechanism. In case that more security is required, the credentials can be passed from the application as second parameter. Furthermore also the "translation" of the messages coming from the API into the domain specific messages and back is very simple in this example. The API-message is just wrapped into the "body" field of the surrounding JSON. In case of more complex domain protocols, this needs more special and complex coding. Because network actions take a certain amount of time and are performed asynchronously, the connect method is callback-based. The 3rd parameter is the callback that will be invoked, if the connect method finished. The developer MUST ensure, that a second invocation of connect() on an already connected M. Stub returns immediately and does not cause a second connection to be established.

```
MessagingStub_Domain.prototype.connect =
function(ownRtcIdentity, credentials, callbackFunction) {
// ...
};
```

Send message function

The sendMessage() method plays a central role. All outgoing messages must be translated to the domain specific protocol and then be send via the connectivity that was established in the connect() method before. The code attached in the appendix C.3.2 shows an example of a sendMessage() method for a simple WebSocket based domain.

```
MessagingStub.prototype.sendMessage = function(message) {
// ...
};
```

Disconnect function

The disconnect() method is for closing and cleaning up the communication channel that was established in the connect() method before. Any signalling that is potentially required by the MS of the certain domain to cleanly shutdown the communication should be done here. The code attached in the appendix C.3.3 shows an example of a disconnect() method for a simple WebSocket based domain.

```
MessagingStub.prototype.disconnect = function() {
// ...
};
```

5.4.1 Naming convention for a MessagingStub

The M. Stub is the subject for on-the-fly download and dynamic instantiation during runtime of the communication application. In order to allow the identification of the Javascript object that implements the M. Stub for a certain domain after the corresponding piece of code has been downloaded, the M. Stub implementation MUST adhere to the following naming convention. Each MessagingStub is identified by an unique download URL, for instance https://domain_host:port/stubs/MessagingStub_SimpleWebSocket.js. The Convention is that the Javascript object that implements the M. Stub MUST be named according to the file-name part of the download url without the trailing ".js". That means, the name of the M. Stub for the download URL above must be "MessagingStub_SimpleWebSocket".

Chapter 6

Validation

This chapter presents the results obtained during the validation of the interoperability platform developed.

In order to do so, the first section describes the test environments to have a deep understanding of the setup. In the last sections the results are presented and evaluated.

The evaluation of the results gives information on interoperability and service delivery for the two signalling platforms analyzed (Web Centric and IP Multimedia Subsystem (IMS)) which will be used in the next chapter to extract conclusions and recommendations.

6.1 Test environments

In order to deploy and test the systems, the infrastructure and services provided by the University of Patras were used. These services were provided as part of the assets for the Webrtc interOperability tested in coNtradictive DEployment scenaRios (WONDER) project in which the work for this thesis was developed.

As mentioned in the previous chapters, the framework developed for this thesis was tested against four domains implemented and deployed in the scope of the WONDER project by its partners, Deutsche Telekom (DT) and Portugal Telecom (PT). The Deutsche Telekom (DT) Web domain was entirely implemented and deployed by the author of this thesis as part of the DT team in the WONDER project.

The test environments for each one of the four domains tested in the experiment were setup as shown in the following subsections.

6.1.1 Deutsche Telekom's Web Centric Test Environment

The DT Web Centric environment is based on a simple Messaging Server, implemented in Node.JS. The client nodes have to connect via an open WebSocket to the Messaging Server. All session exchange is performed inside the JSON syntax. The Idp Server allows identity resolving, including the resource where the Messaging Server is provided. The session establishment and routing is processed by the Messaging Server.

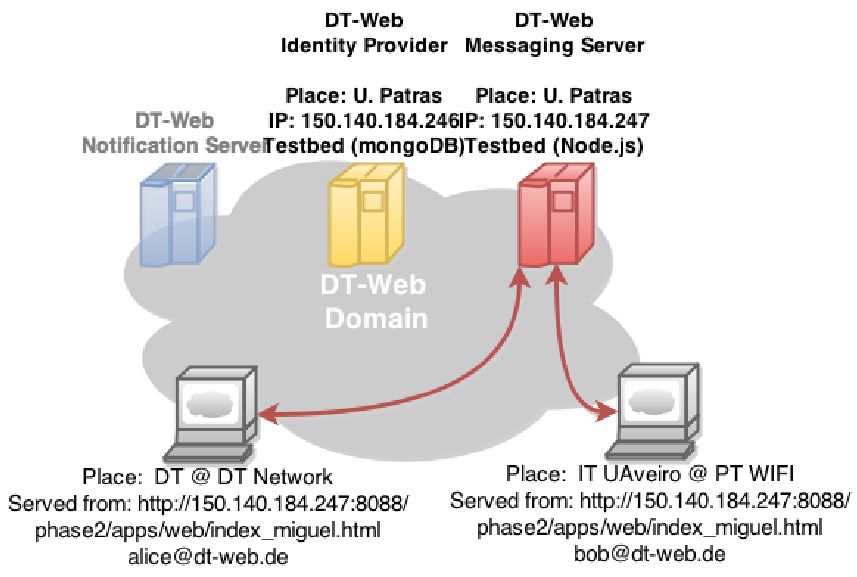


Figure 6.1: DT Web Centric Test Environment

6.1.2 Deutsche Telekom's IMS Test Environment

For this domain, we have to distinguish between intradomain and interdomain calls setup, as in this case Messaging Server (MS) and Client Manager (CM) are not the same entity as in the other domains.

Intradomain calls

The IMS intra-domain environment does not require the provided MessagingServer since the interchanging peers are inside the same domain. For the communication between browser and the DT-IMS CM gateway, a local messaging stub is used which translates the messages produced by the API into the JSON notation of the IMS CM. Resolving of the identities is performed by the Identity Provider (IDP).

Interdomain calls

In this case, the communication with the IMS core will be via the DT-IMS CM gateway for the messages from/to the DT-IMS domain clients and via the DT-IMS MS for the clients from other domains.

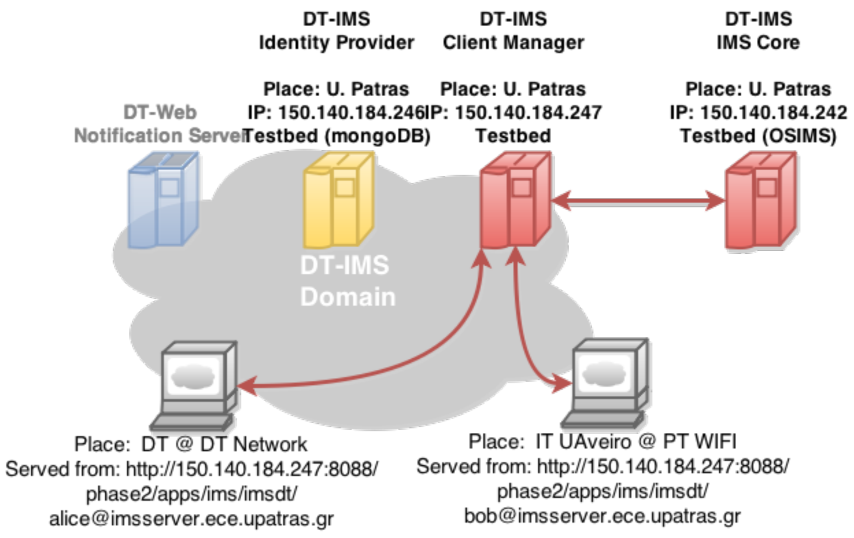


Figure 6.2: DT IMS Centric Intradomain Test Environment

Note that while CM is only a gateway that extends the functionality of the Messaging Stub (M. Stub) translating the API messages to SIP compliant messages, the MS works as a Application Server (AS) inside the IMS core, giving extended functionality needed for interoperability with other domain's clients.

The Figure 6.3 shows an example of an interdomain call between DT-IMS and Portugal Telecom (PT)-web domains.

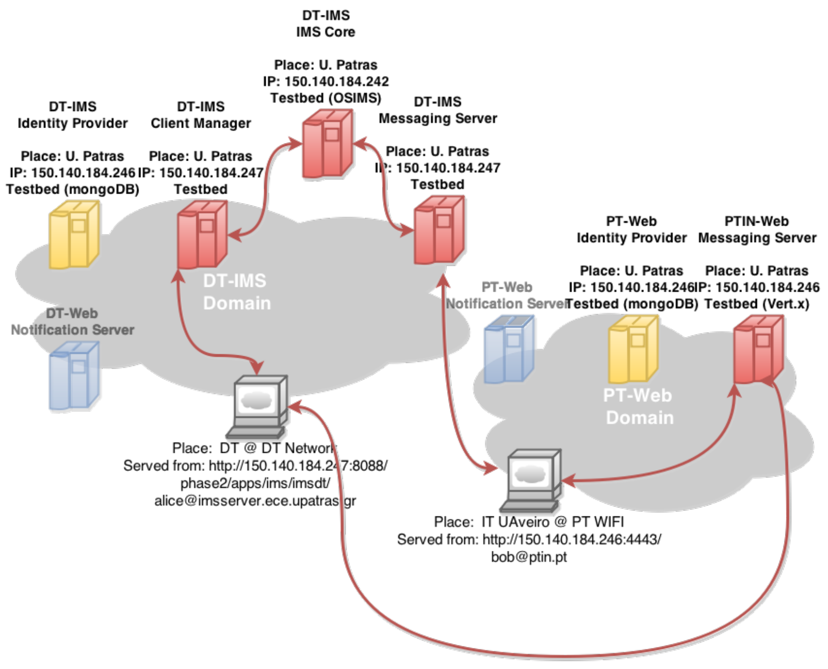


Figure 6.3: DT IMS Centric - PT Web Centric Interdomain Test Environment

OSIMS - Open Source IMS experimentation platform

For this domain, the IMS core platform available on this testbed was the Open Source IMS core.

The OSIMS testbed is depicted in the figure 6.4. The core of the testbed is based on the Open Source IMS core of OpenIMS found at http://www.openimscore.org/. It also provides support for installation of own extensions.

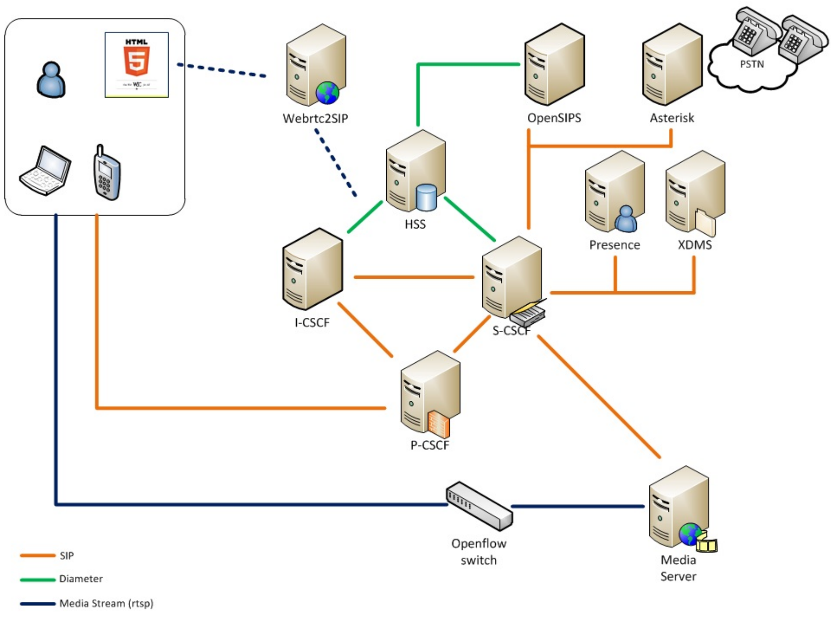


Figure 6.4: OSIMS - Open Source IMS experimentation platform

6.1.3 Portugal Telecom's Web Centric Test Environment

The PT Web Centric environment is based on a simple Messaging Server, implemented in Vert.x and deployed as shown in the Figure 6.5.

6.1.4 Portugal Telecom's SIP Centric Test Environment

The PT Web Centric environment is based on a simple Messaging Server, implemented with WebRTC2SIP against an Asterisk SIP Server and deployed as shown in the Figure 6.6.

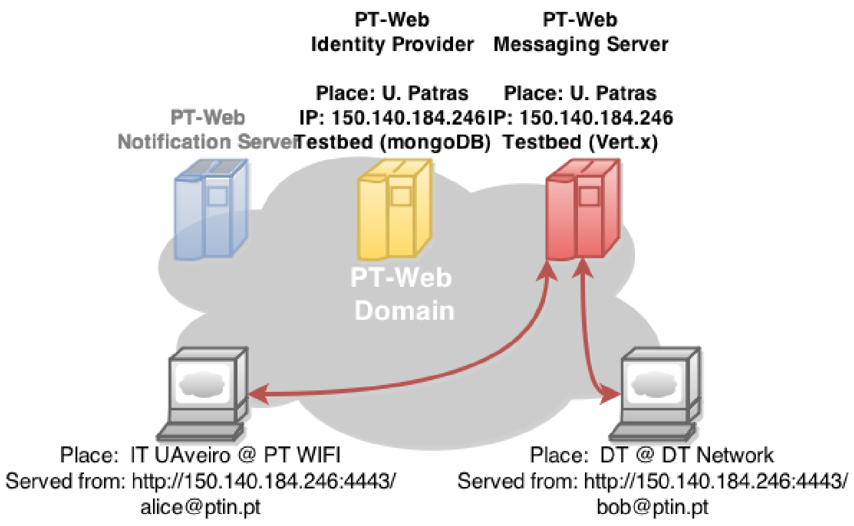


Figure 6.5: PT Web Centric Intradomain Test Environment

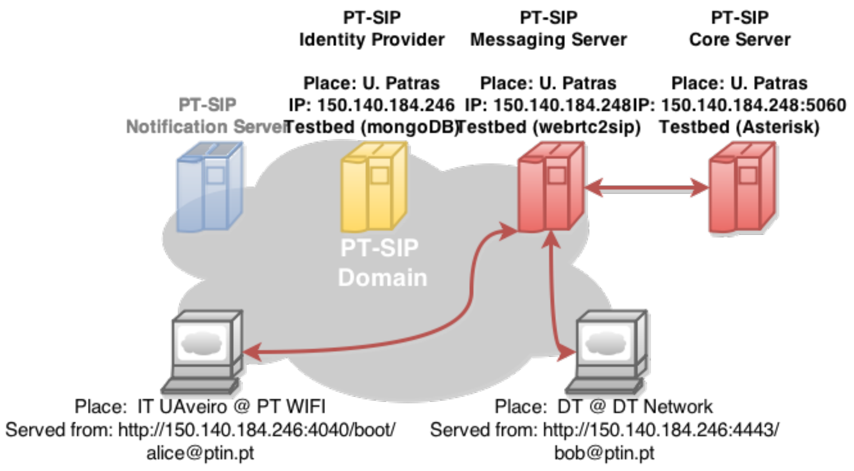


Figure 6.6: PT SIP Centric Intradomain Test Environment

6.2 Results

6.2.1 Intra-domain Experimentations

The intra-domain experimentations results are summarised in the Table 6.1 where the rows represent each one of the tests performed and the columns the domain in which the features were tested. The legend for the colors used in Table 6.1 can be checked in Table 6.9

User Registration

The User Registration use case was successfully experimented in all domains. For simplification purposes a similar RTC Identity syntax was used based on Web URI: username@domain.

Basic Intra-domain A/V Conversation

Audio and Video conversation were successfully established in all domains and in different network conditions e.g. Web Real Time Communication (WebRTC) clients behind firewalls were able to establish WebRTC peer connections by using STUN and TURN servers deployed in the University of Patras Testbed.

Chat-only Conversation

Chat conversations carried on top of Web RTC data channel were successfully established in all domains and also in different network conditions.

A/V Conversations enriched with Chat, File Transfesrs and Screen Share

Enriched Conversations featuring Audio, Video and Chat functionalities were successfully experimented in all domains. Richer conversations with File Transfer and Screen Sharing were experimented in Web centric domains but not in IMS and SIP domains due to lack of available resources. However it is estimated that the effort needed to have File Transfer and Screen Sharing successfully experimented in IMS and SIP domains would be minimum.

Multiparty fully meshed with Hosting

In this experimentation all peers have direct media and data streams established with all remaining peers and a single Hosting MS is used i.e. all peers have a signaling channel established with the same MS. A more detailed explanation of this topology can be found in the Section 4.2.2 and pictured in Figure 4.5.

The tests were successful performed in PT Web centric domain and DT Web centric domain for Enriched Conversations featuring Audio, Video, Chat, File Sharing and Screen Sharing. For IMS and SIP based domains the tests were not performed since the algorithm used would imply very high effort and probably it would not work with standard IMS endpoints. An alternative to this algorithm that may work with IMS and SIP domains is described as Further Work in Section B.2.

MCU based Topology with Hosting

In this experimentation all peers have media and data streams established with a central media server that mixes and distributes streams among the peers, and a single Hosting MS is used i.e. all peers have a signaling channel established with the same MS. Doubango Telepresence [32] system was the MCU chosen for the experiments. A more detailed explanation of this topology can be found in Section 4.2.2 and pictured in Figure 4.7.

The tests were successfully performed in all domains. Since this experimentation was supported on a SIP based Media Server it also implies the usage of the SIP MS for the other domains. It should also be noted that only stream based features like audio, video and screen sharing were tested since data channel based features (Chat and File Sharing) are not supported in the Media Gateway nor in the Media Server.

	DT Web	PT Web	DT IMS	PT SIP
User Registration	░	░	░	░
Basic A/V Conversation	░	░	░	░
Chat	░	░	░	░
A/V and Chat	░	░	░	░
Rich Conversations	░	░	▓	▓
Fully Meshed A/V + Chat Conversation	░	░	█	█
Fully Meshed Rich Conversation	░	░	█	█
MCU based Multiparty A/V Conversation	░	░	░	░

Table 6.1: Summary of Intra-domain experimentations. Legend in Table 6.9

6.2.2 Inter-domain Experimentations

Inter-domain experimentations were conducted in pairs of different domains among the four that were previously introduced.

In general, basic two party Audio and Video conversations were experimented between any combination of pairs among the four domains with no major issue (Table 6.2), therefore successfully demonstrating the signalling on-the-fly concept. Enriched two party conversations with Chat, File Sharing and Screen Sharing (Table 6.4) were successfully tested in Web centric domains including the Conversation updates feature where for example the conversation is open with only chat and then it can be updated to also support Audio and Video. For IMS and SIP based domains only audio, video and chat (Table 6.3) were experimented including the Conversation updates feature where the conversation is open with only chat and then it is updated to also support Audio and Video.

Inter-domain Multiparty enriched Conversations were experimented in two network topologies:

Fully Meshed topology with Hosting

The tests were successful performed in PT Web centric domain and DT Web centric domain for Enriched Conversations featuring Audio, Video, Chat, File Sharing and Screen Sharing (Table 6.5, Table 6.7). For IMS and SIP based domains the tests were not performed since the algorithm used would imply very high effort and probably it would not work with standard IMS endpoints. An alternative algorithm that may work with IMS and SIP domains is described as Further Work in Section B.2.

MCU based Topology with Hosting

The tests were successfully performed in all domains (Table 6.6). Similar to intra-domain tests only stream based features like audio, video and screen sharing were tested since data channel based features (Chat and File Sharing) are not supported in the Media Gateway nor in the Media Server (Table 6.8).

	DT Web Centric	PT Web Centric	DT IMS Centric	PT SIP Centric
DT Web Centric	—	▨	▨	▨
PT Web Centric	▨	—	▨	▨
DT IMS Centric	▨	▨	—	▨
PT SIP Centric	▨	▨	▨	—

Table 6.2: Two party Inter-domain Basic A/V experimentations. Legend in Table 6.9

	DT Web Centric	PT Web Centric	DT IMS Centric	PT SIP Centric
DT Web Centric	—	▨	▨	▨
PT Web Centric	▨	—	▨	▨
DT IMS Centric	▨	▨	—	▨
PT SIP Centric	▨	▨	▨	—

Table 6.3: Two Party Inter-domain AV plus Chat Conversation experimentations. Legend in Table 6.9

	DT Web Centric	PT Web Centric	DT IMS Centric	PT SIP Centric
DT Web Centric	—	░	░	░
PT Web Centric	░	—	░	░
DT IMS Centric	░	░	—	░
PT SIP Centric	░	░	░	—

Table 6.4: Two Party Inter-domain Rich Conversation experimentations. Legend in Table 6.9

	DT Web Centric	PT Web Centric	DT IMS Centric	PT SIP Centric
DT Web Centric	—	▓	█	█
PT Web Centric	░	—	█	█
DT IMS Centric	█	█	—	█
PT SIP Centric	█	█	█	—

Table 6.5: Multi-party Inter-domain Basic AV experimentations in Mesh Topology. Legend in Table 6.9

	DT Web Centric	PT Web Centric	DT IMS Centric	PT SIP Centric
DT Web Centric	—	░	░	░
PT Web Centric	░	—	░	░
DT IMS Centric	░	░	—	░
PT SIP Centric	░	░	░	—

Table 6.6: Multi-party Inter-domain Basic AV experimentations in MCU based Topology. Legend in Table 6.9

	DT Web Centric	PT Web Centric	DT IMS Centric	PT SIP Centric
DT Web Centric	—	▓	█	█
PT Web Centric	░	—	█	█
DT IMS Centric	█	█	—	█
PT SIP Centric	█	█	█	—

Table 6.7: Multi-party Inter-domain Rich Conversation experimentations in Mesh Topology. Legend in Table 6.9

	DT Web Centric	PT Web Centric	DT IMS Centric	PT SIP Centric
DT Web Centric	—	■	■	■
PT Web Centric	■	—	■	■
DT IMS Centric	■	■	—	■
PT SIP Centric	■	■	■	—

Table 6.8: Multi-party Inter-domain Rich Conversation experimentations in MCU based Topology. Legend in Table 6.9

— Not applicable	▒ Fully compatible	▓ Minimum effort needed (days)	■ High effort needed(months)

Table 6.9: Legend for Tables 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7 and 6.8.

6.3 Evaluation

6.3.1 Interoperability

In general inter-domain experimentations were very successful, demonstrating that the signalling on-the-fly concept can be used to enable seamless interoperability between any WebRTC domains with no use of Network to Network Interface (NNI) standard protocols. A standard and protocol-agnostic Javascript API, like the API described in this thesis should be used instead, promoting portability of Applications among different back-end solutions. Such approach, also benefits service providers by minimising dependencies between Applications and back-end vendors. Until now, one of the rationales to use IMS based back-end solutions was the need to have NNI standard interfaces based on SIP to ensure full interoperability between different Service Provider domains. The successful demonstration of the signalling on-the-fly concept also means this rational is not valid anymore. At the end this means a web centric delivery approach using more agile and simpler architectures is feasible and paves the way for a future Web centric standard Service Architecture as an alternative to IMS.

6.3.2 Service delivery (Web centric vs IMS centric)

Looking into the summary experimentation results tables from Section 6.2.1 and Section 6.2.2 we may conclude Web centric delivery approach had more successful results than the IMS centric. This result can be seen as a surprise since IMS is a mature architecture with a large set of services available, while WebRTC is still in very early stages (not a standard yet). In reality, the experimentation developed in this thesis didn't take much advantage of existing services namely

Presence and XDMS services due to the amount of integration effort it would demand. Nevertheless, this also indicates how IMS option implies further integration efforts when compared with the Web centric option.

Chapter 7

Conclusions and further work

This chapter concludes the thesis with a summary of the project, reviewing the final status of the objectives presented in the introduction, some recommendations for the adopters of Web Real Time Communication (WebRTC) technology, either application developers, content providers or service providers. In the Further Work section, some proposals for future implementations and projects are discussed.

7.1 Conclusions

The development and experimentations carried out in this thesis covered the goals outlined for the project in designing, developing and testing a framework for WebRTC applications that allows interoperability between domains with different implementations. The work performed for this thesis included: (1) analysis of the state of the art, (2) the design of the interoperability mechanisms and architecture which constitute the framework, (3) the implementation of an API that integrated the client-side elements of the framework, (4) the implementation of a proof-of-concept domain that integrated the interoperability mechanisms on the server-side, (5) validation of the platform among different domains (including the proof-of-concept domain developed for this thesis), (6) documentation of the work performed.

The results extracted from this work are also intended to answer the main questions about whether to use Web centric or IP Multimedia Subsystem (IMS) solutions from the validation performed among two different web centric domains and two IMS and SIP based domains.

The results of this work show how the signalling gap that WebRTC leaves can be filled with a solution featuring signalling on-the-fly that can be adapted to the existing signalling infrastructures, such as IMS, to enable seamless interoperability between different WebRTC Service Provider domains to avoid the usage of standard Network to Network Interface (NNI) protocols. The problem on how to address peer users that reside in another WebRTC service domain was solved by extending the current Identity WebRTC specification and introducing the Identity Provider (IDP) entity.

WebRTC is clearly designed thinking on application development and Over The Top (OTT) services and not as a core element for Communication Service Providers. This idea is confirmed by the tests performed, which compare Web Centric solutions with former IMS/SIP solutions showing that, in general, web centric service delivery option promise to bring more advantages to Service Providers and Developers than IMS/SIP Service Delivery option. For this reason, if Communication Service Providers want to take advantage of the benefits of WebRTC technology, they must evaluate whether to use a Web Centric approach or use their current IMS infrastructure.

- Application developers, OTT providers such as video-on-demand or multimedia conference services, and vertical applications e.g. Education or Healtcare, are advised to use a Web Centric signalling solution which can be easily made interoperable with services from different domains, as proved in this work.

- Communication Service Providers that have already deployed a IMS network infrastructure featuring a rich set of communication services like MMTEL, VoLTE, or Rich Communication Services/Joyn (RCS/Joyn) are recommended in the short term to implement WebRTC services over this infrastructure, even if the convergence with the existing technologies such as SIP or PSTN is not mature enough yet.

- Communication Service Providers without a IMS network infrastructure are advised to go for a Web Centric approach in spite of deploy an IMS infrastructure to offer WebRTC services.

7.2 Further work

This work covered a proof-of-concept solution for interoperability in WebRTC. However, it leaves some challenging topics which are open for further research and implementation.

Firstly, it is important to highlight the fact that this thesis is the product of a research work, whose main goal wasn't to be put into production. For this reason some important functionalities, such as call blocking, charging policies, etc. are left as future work.

From the current implementation, points that need further work are the implementation of the topology without hosting; the addressing of the IDP for each domain, which is now hardcoded into the application and should be resolved somehow; a better standardization of the Messaging Stub (M. Stub) implementation, including one M. Stub per technology and customizing it with the variables needed to make it work for a specific domain, making a better use of the namespaces and avoiding the override of functions, increasing security.

In view of the results obtained in the tests, where the domains that didn't support PUBLISH (multicast of messages to all peers in a conversation) and message buffering in the Messaging Server (MS) couldn't make multiparty meshed calls, is evident the need of a multiparty algorithm for this kind of calls that avoids the need of such mechanisms. An alternative algorithm is presented in the Appendix B.2 proposed as further implementation work.

Bibliography

[1] EURESCOM, P2252 - Telco strategic positioning options regarding WebRTC.
URL http://www.eurescom.eu/services/eurescom-study-programme/list-of-eurescom-studies/studies-launched-in-2012/p2252.html

[2] Google Inc., WebRTC.
URL http://www.webrtc.org/

[3] Sam Dutton, WebRTC in the real world: STUN, TURN and signaling.
URL http://www.html5rocks.com/en/tutorials/webrtc/infrastructure

[4] Introducing JSON.
URL http://json.org

[5] Node JS vs Apache PHP benchmark.
URL https://code.google.com/p/node-js-vs-apache-php-benchmark/wiki/Tests

[6] Kurento Website.
URL http://www.kurento.org/

[7] Orca.js Website.
URL http://www.orcajs.org/

[8] Doubango Telecom ®, WebRTC2SIP - Smart SIP and Media Gateway to connect WebRTC endpoints.
URL http://webrtc2sip.org/

[9] Reuters, U.S. seeks trials to test transition to digital phone networks, 2014.
URL http://www.reuters.com/article/2014/01/30/usa-fcc-iptransition-idUSL2N0L414G20140130

[10] Brian Stelter, Amy Chozick, Viewers Start to Embrace Television on Demand, May 2013.
URL `http://www.nytimes.com/2013/05/21/business/media/video-on-demand-viewing-is-gaining-popularity.html?_r=0`

[11] Steven Pemberton, The Future of Web Applications, 2005.
URL `http://www.w3.org/2005/Talks/09-steven-interact/`

[12] Dan York, How WebRTC Will Fundamentally Disrupt Telecom (And Change The Internet).
URL `http://www.disruptivetelephony.com/2012/09/how-webrtc-will-fundamentally-disrupt-telecom-and-change-the-internet.html`

[13] Brent Kelly, Preparing for Disruption with WebRTC.
URL `http://www.nojitter.com/post/240155570/preparing-for-disruption-with-webrtc`

[14] 3GPP, Service requirements for the Internet Protocol (IP) multimedia core network subsystem (IMS), TS 22.228 , release 13, December 2014.
URL `http://www.3gpp.org/ftp/specs/archive/22_series/22.228/22228-d20.zip`

[15] W3C, WebRTC API.
URL `http://w3c.github.io/webrtc-pc/`

[16] W3C, WebRTC Media Capture and Streams API.
URL `http://w3c.github.io/mediacapture-main/`

[17] J. Uberti, C. Jennings, Javascript Session Establishment Protocol.
URL `http://tools.ietf.org/html/draft-ietf-rtcweb-jsep-03#section-3.3`

[18] 3GPP, Overview of 3GPP Release 5 – Summary of all Release 5 Features, 3GPP-ETSI Mobile Competence Centre, 2003.
URL `ftp://www.3gpp.org/tsg_ran/TSG_RAN/TSGR_20/Docs/PDF/RP-030375.pdf`

[19] [3GPP, Service requirements for the Internet Protocol (IP) multimedia core network subsystem (IMS), TS 22.228.
URL `http://www.3gpp.org/DynaReport/22228.htm`

[20] J. Postel, Internet Protocol, Internet RFC 0791, 1981.
URL `https://tools.ietf.org/html/rfc0791`

[21] S. Deering and R. Hinden, Internet Protocol, Version 6 (IPv6) Specification, Internet RFC 2460, 1998.
URL `https://tools.ietf.org/html/rfc2460`

[22] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, SIP: Session Initiation Protocol, Internet RFC 3261, 2002.
URL `https://tools.ietf.org/html/rfc3261`

[23] M. Handley and V. Jacobson, SDP: Session Description Protocol, Internet RFC 2327, 1998.
URL `https://tools.ietf.org/html/rfc2327`

[24] [14] D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, and A. Sastry, Diameter Base Protocol, Internet RFC 3588, 2003.
URL `https://tools.ietf.org/html/rfc3588`

[25] P. Calhoun, J. Loughney, E. Guttman, G. Zorn, J. Arkko, Ericsson, The COPS (Common Open Policy Service) Protocol, Internet RFC 2748, 2000.
URL `https://tools.ietf.org/html/rfc2748`

[26] ITU-T, Gateway control protocol: Version 3, Recommendation H.248, 2013.
URL `http://www.itu.int/rec/T-REC-H.248.1-201303-I/en`

[27] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, RTP: A Transport Protocol for Real-Time Applications, Internet RFC 3550, July 2003.
URL `https://tools.ietf.org/html/rfc3550`

[28] Scott Hanselman, JavaScript is Assembly Language for the Web: Part 2 - Madness or just Insanity?, July 2011.
URL `http://www.hanselman.com/blog/JavaScriptisAssemblyLanguagefortheWebPart2Madnessor`
`aspx`

[29] D. Crockford, The application/json Media Type for JavaScript Object Notation (JSON), Internet RFC 4627, July 2006.
URL `https://tools.ietf.org/html/rfc4627`

[30] Ian Hickson, Robin Berjon, Steve Faulkner, Travis Leithead, Erika Doyle Navara, Edward O'Connor, Silvia Pfeiffer, HTML5. A vocabulary and associated APIs for HTML and XHTML, October 2014.
URL `http://www.w3.org/TR/html5/`

[31] Simon Pieters, HTML5 Differences from HTML4, December 2014.
URL `http://www.w3.org/TR/html5-diff/`

[32] Doubango Telecom ®, Telepresence - the open source SIP TelePresence system.
URL `https://code.google.com/p/telepresence/`

[33] Steffen Drüsedow, Kay Hänsge, Miguel Seijo, Paulo Chainho, Vasco Amaral, Luis Oliveira, WONDER – Assessment Report, April 2014.
URL http://www.ict-openlab.eu/fileadmin/documents/public_deliverables/OpenLab_Deliverable_D4_15.pdf

[34] AppRTC example.
URL https://apprtc.appspot.com/

Appendix A

Budget

This appendix calculates the budget of the work developed in this thesis. The first two sections describe the material resources and project phases. Last two sections calculate the total budget from the costs estimated on materials and human resources described in the previous sections.

A.1 Material Resources

The material resources needed to carry out this thesis are explained below.

All the materials plus the infrastructure and Internet access were provided by Deutsche Telekom Innovation Laboratories (T-Labs) for the Webrtc interOperability tested in coNtradictive DEployment scenaRios (WONDER) project during an Erasmus Placement internship in Berlin.

- Laptop Lenovo G580 for development, testing and documentation writing.
- Display Fujitsu-Siemens B24W as primary display.
- Laptop Fujitsu-Siemens T-series for testing.
- HP 5-port switch for network access.

Additionally, some virtual machines and testbeds were provided by University of Patras (UoP) and Waterford Institute of Technology (WIT) for the WONDER project.

A.2 Project Phases

The work developed for this thesis was divided in one phase where the main task was to analyze the state of the art, two phases where the actual development took place and a final phase where the main task consisted on the documentation of the whole work.

A.2.1 Analysis of the State of the Art

During this phase, the main task was to make a detailed study on the state of the art related to Web Real Time Communication (WebRTC) and different domains for interoperability.
To carry out this phase, it was needed the effort of one person for a duration of 7 days.

A.2.2 Experimentation Phase 1: Basic Interoperability, Rich Conversations

During this phase the main objectives of this thesis were accomplished for the most basic scenarios, such as:

- User registration
- Basic intra-domain A/V Call
- Basic inter-domain A/V Call
- Rich communications (chat)

To carry out this phase, it was needed the effort of one person for total duration of 95 days, as dependencies in the tasks forced concurrency between them.

Specification 1

This task included the discussion and design, for the basic scenarios described above, of the architecture and interoperability mechanisms, the API to create interoperable WebRTC applications, and a web-based signalling domain for testing.

This task needed the effort of one person for a total duration of 3 weeks, which is the equivalent to 15 working days.

Implementation 1

This task consisted on the installation and configuration of the workspace and the implementation of the features that were discussed previously.

This task needed the effort of one person for a total duration of 80 days.

Testbeds update and configuration 1

During this task, the integration, update and configuration of the workspace into the testbeds were performed.

This task needed the effort of one person for a total duration of 5 days.

Tests performance 1

This task consisted on executing the test cases addressed in the specifications for the framework and API created in the previous tasks using the test signalling domain developed and other signalling domains provided.

This task needed the effort of one person for a total duration of 5 days.

A.2.3 Experimentation Phase 2: Multiparty, Identities, Resources

During this phase new functionalities were designed, tested and implemented for the more advanced scenarios, such as:

- Multiparty meshed Call
- Multiparty MCU Call
- Identity resolving
- Resource (e.g. Audio, Video, A/V, Chat) updates during a conversation.

To carry out this phase, it was needed the effort of one person for total duration of 45 days, as dependencies in the tasks forced concurrency between them.

Specification 2

In this task, the main goal was to discuss and design the changes needed to implement the new functionalities in the framework that were already implemented and tested.

This task needed the effort of one person for a total duration of 2 weeks, which is the equivalent to 10 working days.

Implementation 2

This task consisted on the implementation of the features that were discussed in the previous stage.

This task needed the effort of one person for a total duration of 40 days.

Testbeds update and configuration 2

During this task, the integration, update and configuration of the workspace into the testbeds were performed.

This task needed the effort of one person for a total duration of 1 week, which is the equivalent to 5 working days.

Tests performance 2

This task consisted on executing the test cases addressed in the specifications for the framework and API created in the previous tasks using the test signalling domain developed and other signalling domains provided.

This task needed the effort of one person for a total duration of 1 weeks, which is the equivalent to 5 working days.

A.2.4 Documentation

During the last phase, the main task was to write the thesis documenting all the work and the conclusions drawn during its development.

To carry out this phase, it was needed the effort of one person for total duration of 41 days, as dependencies in the tasks forced concurrency between them.

A.3 Material Expenses

Material expenses account for the material needed for the implementation, i.e. two laptop PCs, a desktop monitor and a 5 port switch. All this material belong to T-Labs and its life span is longer that the project duration, so amortization has to be calculated. Laptop PCs are considered to have a life span of 36 months and a life span of 48 months for the rest of the materials. The project was developed and tested using software that doesn't need to be licensed, so no license costs are taken into account.

The amortization calculations take into account a project duration of 7 months.

The results of the material expenses calculations are presented in Table A.1

A.4 Human Resources Expenses

The calculation of the human resources expenses consider the sum of the work hours, which are calculated considering the effort of one engineer for all the tasks and taking into account for the duration the effect of the overlapping tasks. The total number of days needed to complete this thesis is 151 working days, equivalent to 1208h assuming 8 working hours per working day.

The results of the calculation for this expenses are shown in Table A.2

Concept	Units	Cost/Unit	Amortized (%)	Total
Lenovo G580	1	500€	19.4	97€
Fujitsu-Siemens T-series	1	1000€	19.4	194€
Fujitsu-Siemens Display B24W	1	200€	14.6	29€
HP 5-port switch	1	20€	14.6	3€
Total				323€

Table A.1: Material expenses

Category	Working hours	Cost/Hour	Total
Engineer	1208	30€/hour	
Total			36240€

Table A.2: Human Resources expenses

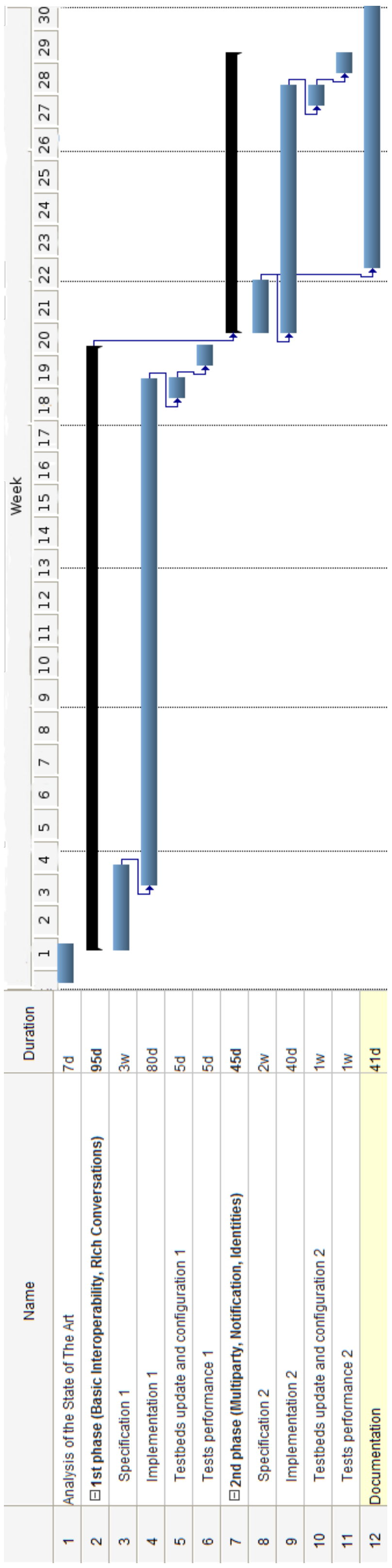
A.5 Total Expenses

The Table A.3 shows the final budget of the project from the expenses calculated before.

Concept	Total
Material Expenses	323€
Human Resources Expenses	36240€
Total	36563€

Table A.3: Total Expenses

The total budget of this project amounts to thirty-six thousand five hundred sixty-three Euros.



	Name	Duration
1	Analysis of the State of The Art	7d
2	⊟ **1st phase (Basic Interoperability, Rich Conversations)**	**95d**
3	Specification 1	3w
4	Implementation 1	80d
5	Testbeds update and configuration 1	5d
6	Tests performance 1	5d
7	⊟ **2nd phase (Multiparty, Notification, Identities)**	**45d**
8	Specification 2	2w
9	Implementation 2	40d
10	Testbeds update and configuration 2	1w
11	Tests performance 2	1w
12	Documentation	41d

Figure A.1: Gannt diagram

Appendix B

Call Establishment Algorithms

B.1 Call Establishment Algorithm for Multiparty

In order to establish a call in a multiparty conversation with hosting where Alice is the host and invites Bob and Carol, the following signalling messages will be exchanged: Note: This explanation supposes that the Client Manager (CM) is integrated with the Messaging Server (MS) (as it happens in most domains). Otherwise, Alice will connect to her CM instead of MS.

Invitation notification (Figures B.1 and B.2)

1. Alice invites Bob and Carol that are subscribed to different domains. The invitations should have different Session Description Protocol (SDP).
2. Alice's MS resolves Bob and Carol's restful notification service server from their Identity Provider (IDP).
3. Alice's MS sends a notification associated to an invitation (which contains the SDP) to Bob and another to Carol.
4. Bob and Carol receive the notification and resolve Alice's Invitation, obtaining Alice's SDP.

Bob accepts invitation (Figure B.3)

5. Bob queries Alice's IDP to resolve the address for Alice's domain Messaging Stub (M. Stub) and downloads it.
6. Bob connects to Alice's MS and sends an ACCEPTED message to Alice containing his SDP answer.
7. Alice receives the ACCEPTED message from Bob.

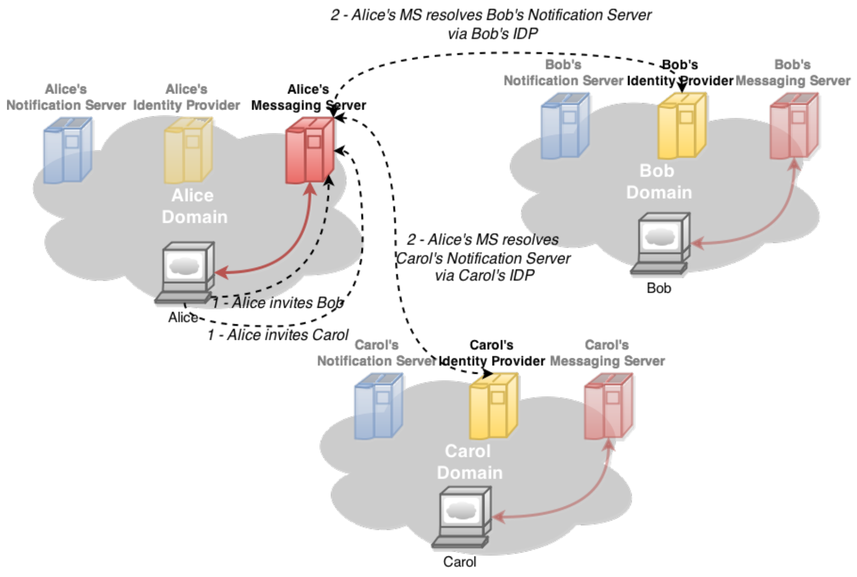

2 - Alice's MS resolves Bob's Notification Server via Bob's IDP
Alice's Notification Server
Alice's Identity Provider
Alice's Messaging Server
Alice Domain
Alice
1 - Alice invites Bob
1 - Alice invites Carol
Bob's Notification Server
Bob's Identity Provider
Bob's Messaging Server
Bob Domain
Bob
2 - Alice's MS resolves Carol's Notification Server via Carol's IDP
Carol's Notification Server
Carol's Identity Provider
Carol's Messaging Server
Carol Domain
Carol

Figure B.1: Call establishment algorithm, step 1 - Multiparty with hosting

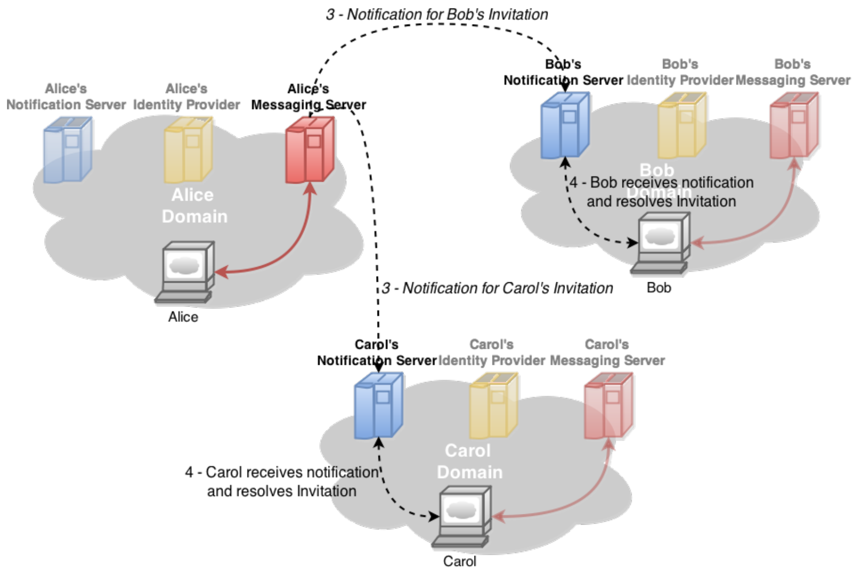

3 - Notification for Bob's Invitation
Alice's Notification Server
Alice's Identity Provider
Alice's Messaging Server
Alice Domain
Alice
Bob's Notification Server
Bob's Identity Provider
Bob's Messaging Server
Bob Domain
4 - Bob receives notification and resolves Invitation
Bob
3 - Notification for Carol's Invitation
Carol's Notification Server
Carol's Identity Provider
Carol's Messaging Server
Carol Domain
4 - Carol receives notification and resolves Invitation
Carol

Figure B.2: Call establishment algorithm, step 2 - Multiparty with hosting

Alice publishes that Bob has Accepted (Figure B.4)

8. The organizer (Alice) publishes on her own MS Bob's ACCEPTED without SDP and a list of connected participants to all the other participants of the conversation. For the participants that are not connected yet to Alice's MS, this message will be cached there.

9. The message will be received by all the clients connected to Alice's MS (in this case, Bob). They will check the list of connected peers and if they are not in the list, they will connect to the peer that sent the original accepted.

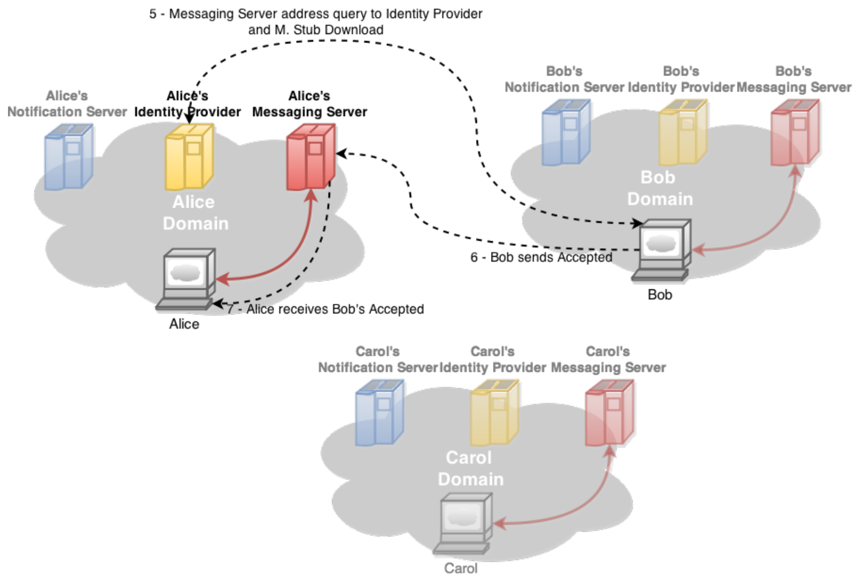

5 - Messaging Server address query to Identity Provider and M. Stub Download
Alice's Notification Server
Alice's Identity Provider
Alice's Messaging Server
Alice Domain
Alice
7 - Alice receives Bob's Accepted
Bob's Notification Server
Bob's Identity Provider
Bob's Messaging Server
Bob Domain
6 - Bob sends Accepted
Bob
Carol's Notification Server
Carol's Identity Provider
Carol's Messaging Server
Carol Domain
Carol

Figure B.3: Call establishment algorithm, step 3 - Multiparty with hosting

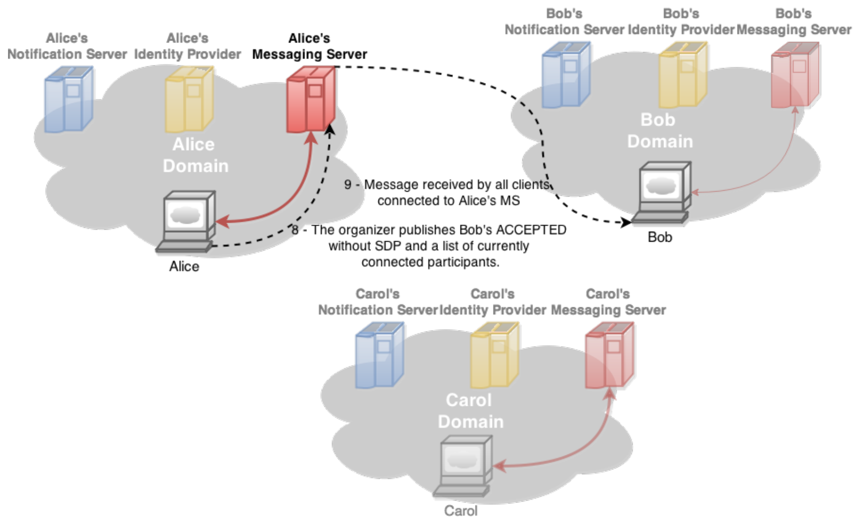

Alice's Notification Server
Alice's Identity Provider
Alice's Messaging Server
Alice Domain
Alice
9 - Message received by all clients connected to Alice's MS
8 - The organizer publishes Bob's ACCEPTED without SDP and a list of currently connected participants.
Bob's Notification Server
Bob's Identity Provider
Bob's Messaging Server
Bob Domain
Bob
Carol's Notification Server
Carol's Identity Provider
Carol's Messaging Server
Carol Domain
Carol

Figure B.4: Call establishment algorithm, step 4 - Multiparty with hosting

Carol Accepts Invitation (Figures B.5 and B.6)

10. Carol queries Alice's IDP to resolve the address for Alice's domain M. Stub and downloads it.

11. Carol connects to Alice's domain MS and sends the ACCEPTED message with the SDP.

12. Carol receives Bob's ACCEPTED message without SDP previously published by Alice that was in the MS cache. As Carol is not in the connected list, it will trigger the connection

from Carol to all the participants in the list that she is not connected to already (in this case Bob).

13. The organizer (Alice) publish Carol's ACCEPTED without SDP and a list of connected participants to all the other participants of the conversation. For the participants that are not connected yet, this message will be cached.

    When Bob receives Carol's ACCEPTED without SDP, he will find himself on the connected list so he will ignore the message.

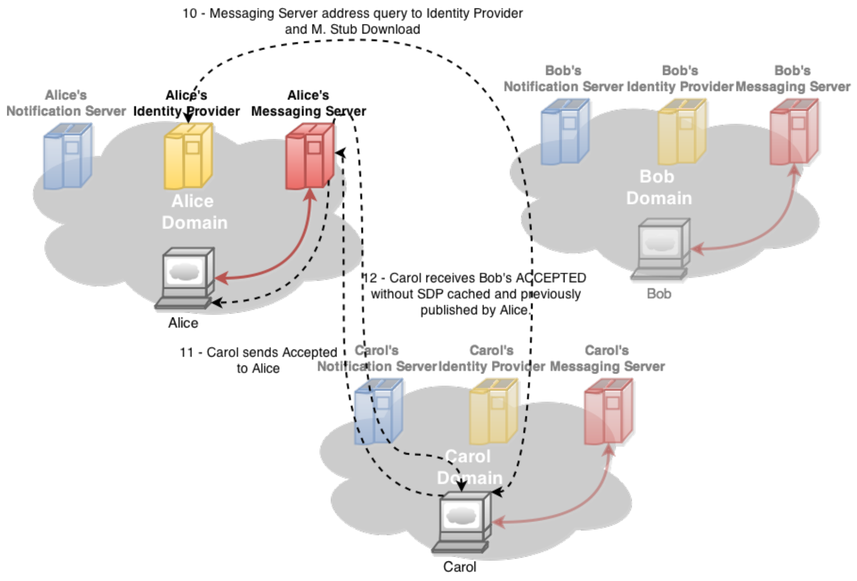

10 - Messaging Server address query to Identity Provider and M. Stub Download
Alice's Notification Server
Alice's Identity Provider
Alice's Messaging Server
Alice Domain
Alice
Bob's Notification Server
Bob's Identity Provider
Bob's Messaging Server
Bob Domain
Bob
12 - Carol receives Bob's ACCEPTED without SDP cached and previously published by Alice
11 - Carol sends Accepted to Alice
Carol's Notification Server
Carol's Identity Provider
Carol's Messaging Server
Carol Domain
Carol

Figure B.5: Call establishment algorithm, step 5 - Multiparty with hosting

Interconnection between peers - Carol invites Bob (Figure B.7)

14. Triggered when Carol receives Bob's ACCEPTED and after checking she is not on the connected list (step 12), Carol invites Bob through Alice's MS.

15. Bob receives Carol's invitation and automatically accepts it.

16. Carol receives Bob's accepted and the connection between them gets established.

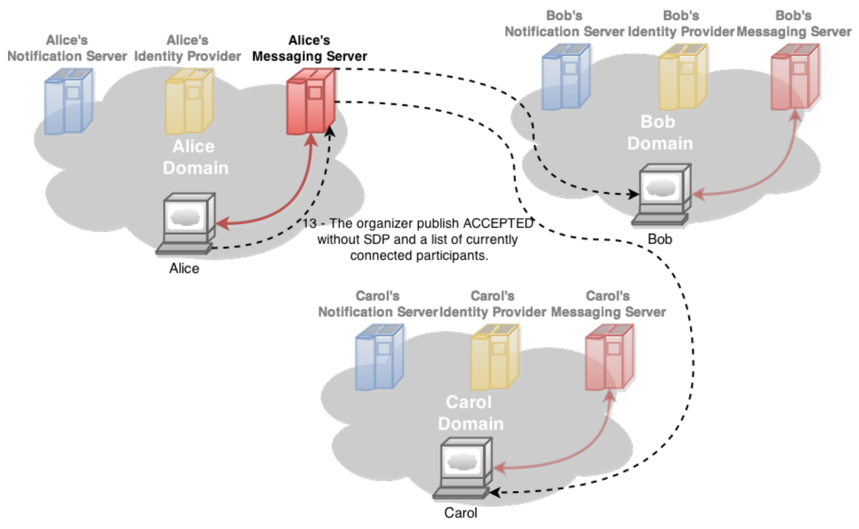

Alice's Notification Server
Alice's Identity Provider
Alice's Messaging Server
Alice Domain
Alice
Bob's Notification Server
Bob's Identity Provider
Bob's Messaging Server
Bob Domain
Bob
13 - The organizer publish ACCEPTED without SDP and a list of currently connected participants.
Carol's Notification Server
Carol's Identity Provider
Carol's Messaging Server
Carol Domain
Carol

Figure B.6: Call establishment algorithm, step 6 - Multiparty with hosting

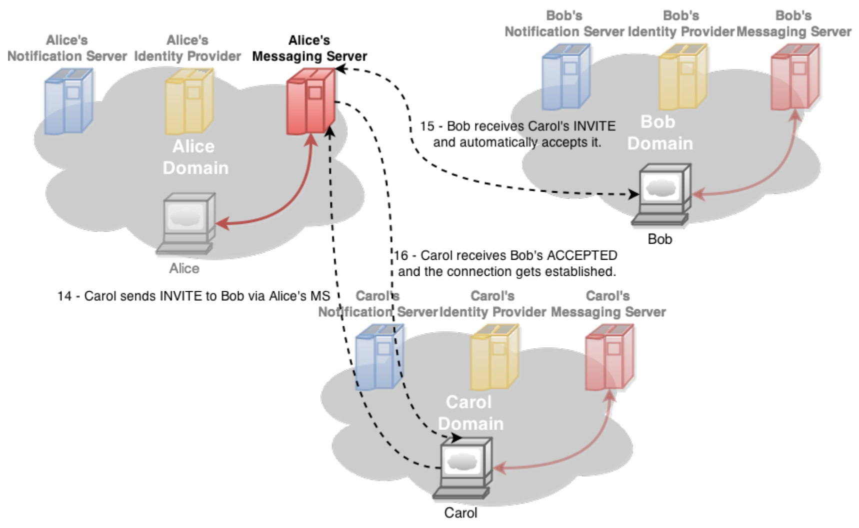

Alice's Notification Server
Alice's Identity Provider
Alice's Messaging Server
Alice Domain
Alice
Bob's Notification Server
Bob's Identity Provider
Bob's Messaging Server
Bob Domain
Bob
15 - Bob receives Carol's INVITE and automatically accepts it.
16 - Carol receives Bob's ACCEPTED and the connection gets established.
14 - Carol sends INVITE to Bob via Alice's MS
Carol's Notification Server
Carol's Identity Provider
Carol's Messaging Server
Carol Domain
Carol

Figure B.7: Call establishment algorithm, step 7 - Multiparty with hosting

B.2 Alternative Call Establishment Algorithm for Multiparty

This new algorithm, which is proposed as further work, avoids the need of a PUBLISH feature in the Messaging Server (MS), increasing the compatibility with domains that don't support this feature. In order to establish a call in a multiparty conversation with hosting where Alice is the host and invites Bob and Carol, the following signaling messages will be exchanged: Note: This explanation supposes that the Client Manager (CM) is integrated with the MS (as it happens in most domains). Otherwise, Alice will connect to her CM instead of MS.

Invitation notification (Figures B.8 and B.9)

1. Alice invites Bob and Carol that are subscribed to different domains. The invitations should have different Session Description Protocol (SDP).

2. Alice's MS resolves Bob and Carol's restful notification service server from their Identity Provider (IDP).

3. Alice's MS sends a notification associated to an invitation (which contains the SDP) to Bob and another to Carol.

4. Bob and Carol receive the notification and resolve Alice's Invitation, obtaining Alice's SDP.

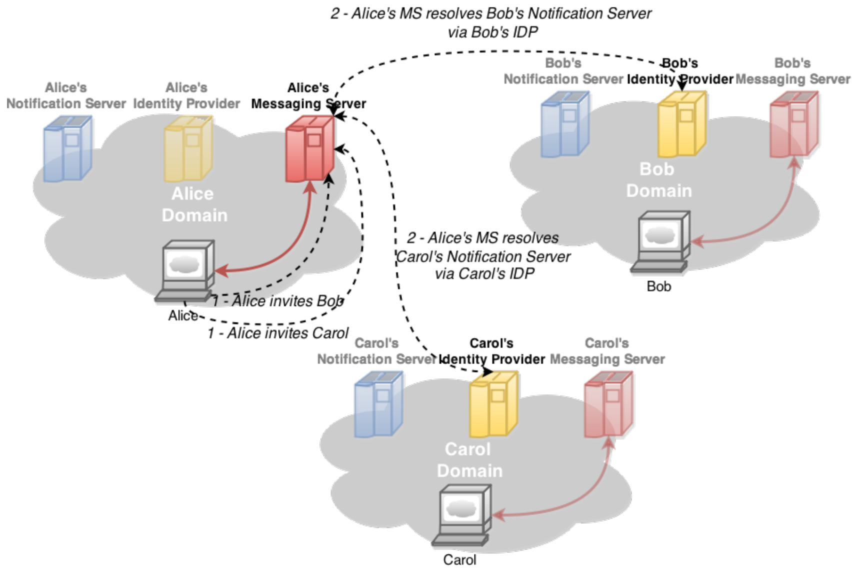

2 - Alice's MS resolves Bob's Notification Server via Bob's IDP
Alice's Notification Server
Alice's Identity Provider
Alice's Messaging Server
Alice Domain
Bob's Notification Server
Bob's Identity Provider
Bob's Messaging Server
Bob Domain
Bob
2 - Alice's MS resolves Carol's Notification Server via Carol's IDP
1 - Alice invites Bob
Alice
1 - Alice invites Carol
Carol's Notification Server
Carol's Identity Provider
Carol's Messaging Server
Carol Domain
Carol

Figure B.8: Call establishment algorithm, step 1 - Multiparty with hosting

Bob accepts invitation (Figure B.10)

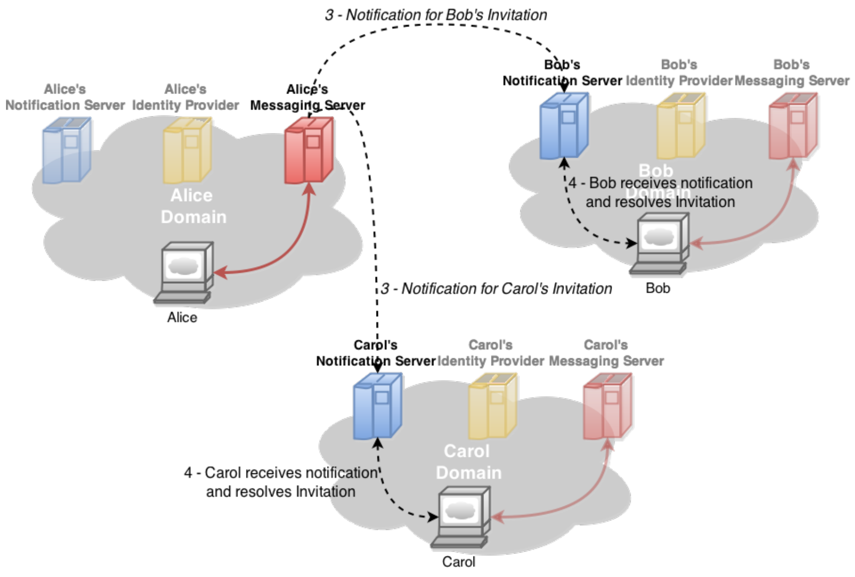

3 - Notification for Bob's Invitation
Alice's Notification Server
Alice's Identity Provider
Alice's Messaging Server
Alice Domain
Alice
Bob's Notification Server
Bob's Identity Provider
Bob's Messaging Server
Bob Domain
4 - Bob receives notification and resolves Invitation
Bob
3 - Notification for Carol's Invitation
Carol's Notification Server
Carol's Identity Provider
Carol's Messaging Server
Carol Domain
4 - Carol receives notification and resolves Invitation
Carol

Figure B.9: Call establishment algorithm, step 2 - Multiparty with hosting

5. Bob queries Alice's IDP to resolve the address for Alice's domain Messaging Stub (M. Stub) and downloads it.

6. Bob connects to Alice's MS and sends an ACCEPTED message to Alice containing his SDP answer.

7. Alice receives the ACCEPTED message from Bob.

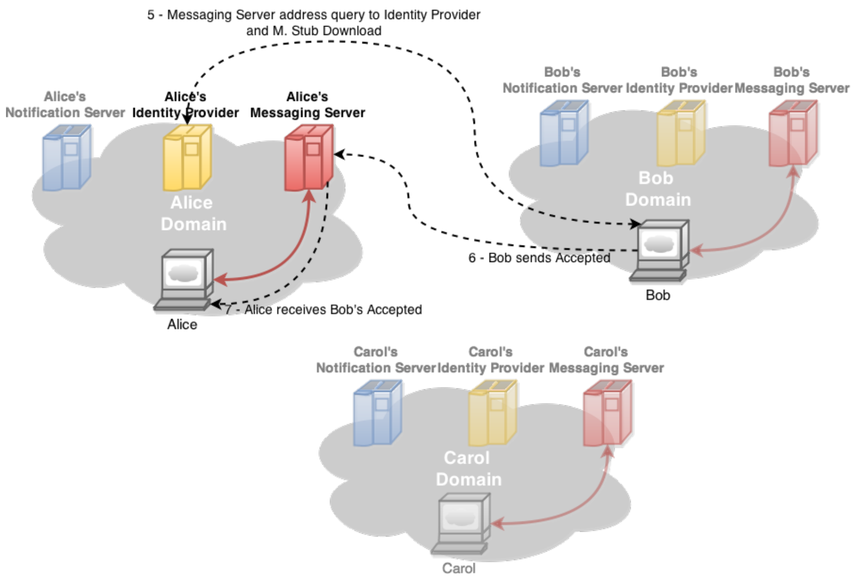

5 - Messaging Server address query to Identity Provider and M. Stub Download
Alice's Notification Server
Alice's Identity Provider
Alice's Messaging Server
Alice Domain
Alice
7 - Alice receives Bob's Accepted
Bob's Notification Server
Bob's Identity Provider
Bob's Messaging Server
Bob Domain
6 - Bob sends Accepted
Bob
Carol's Notification Server
Carol's Identity Provider
Carol's Messaging Server
Carol Domain
Carol

Figure B.10: Call establishment algorithm, step 3 - Multiparty with hosting

Alice answers Bob with the list of connected participants (Figure B.11)

8. The organizer (Alice) sends through her own MS Bob's ACCEPTED without SDP and a list of currently connected participants to Bob (empty in this case as Bob is the first participant to connect).

9. Bob will check the list of connected peers and connect to every one of them. (In this case Bob will do nothing as the list is empty)

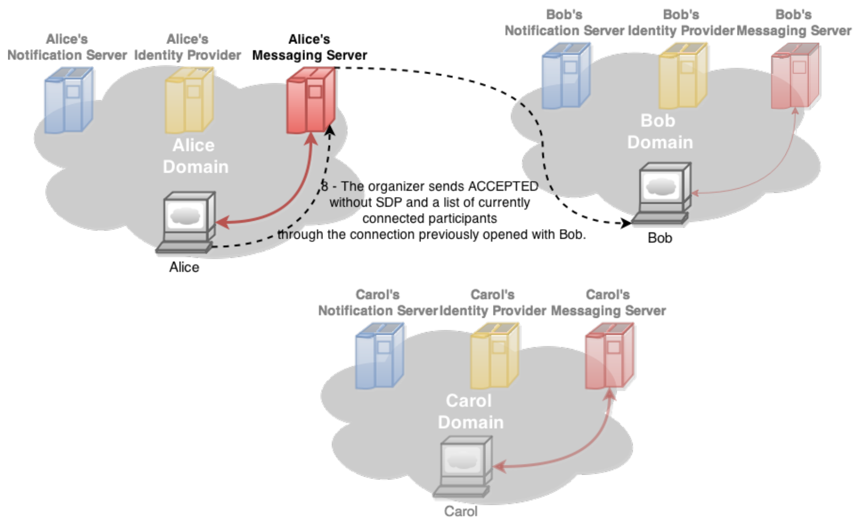

Alice's Notification Server
Alice's Identity Provider
Alice's Messaging Server
Alice Domain
Alice
8 - The organizer sends ACCEPTED without SDP and a list of currently connected participants through the connection previously opened with Bob.
Bob's Notification Server
Bob's Identity Provider
Bob's Messaging Server
Bob Domain
Bob
Carol's Notification Server
Carol's Identity Provider
Carol's Messaging Server
Carol Domain
Carol

Figure B.11: Call establishment algorithm, step 4 - Multiparty with hosting

Carol Accepts Invitation (Figure B.12)

10. Carol queries Alice's IDP to resolve the address for Alice's domain M. Stub and downloads it.

11. Carol connects to Alice's domain MS and sends the ACCEPTED message with the SDP.

12. Alice receives the ACCEPTED message from Carol.

Alice answers Carol with the list of connected participants (Figure B.13)

13. The organizer (Alice) sends through her own MS Carol's ACCEPTED without SDP and a list of currently connected participants to Carol (containing Bob in this case as he is the only one connected).

14. Carol will check the list of connected peers and connect to every one of them. (In this case to Bob).

Interconnection between peers - Carol invites Bob (Figure B.14)

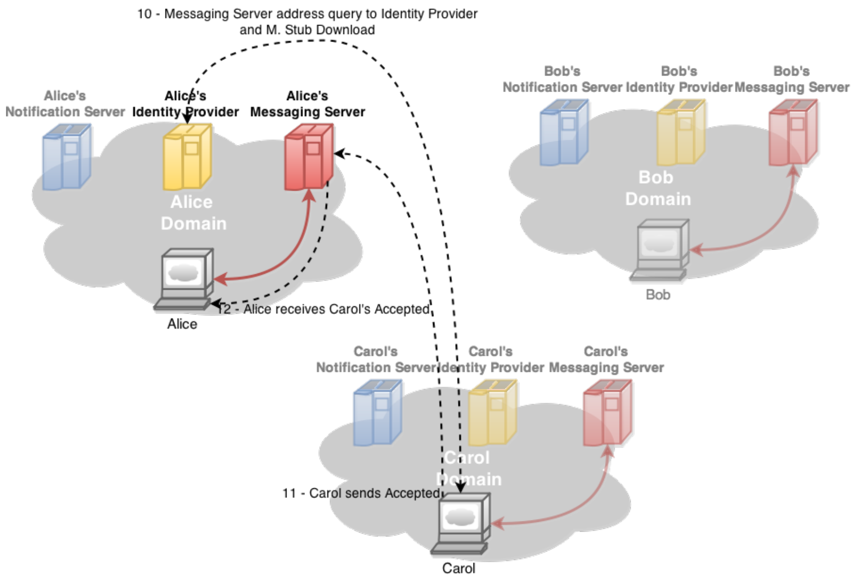

10 - Messaging Server address query to Identity Provider and M. Stub Download
Alice's Notification Server
Alice's Identity Provider
Alice's Messaging Server
Alice Domain
Alice
12 - Alice receives Carol's Accepted
Bob's Notification Server
Bob's Identity Provider
Bob's Messaging Server
Bob Domain
Bob
Carol's Notification Server
Carol's Identity Provider
Carol's Messaging Server
Carol Domain
11 - Carol sends Accepted
Carol

Figure B.12: Call establishment algorithm, step 5 - Multiparty with hosting

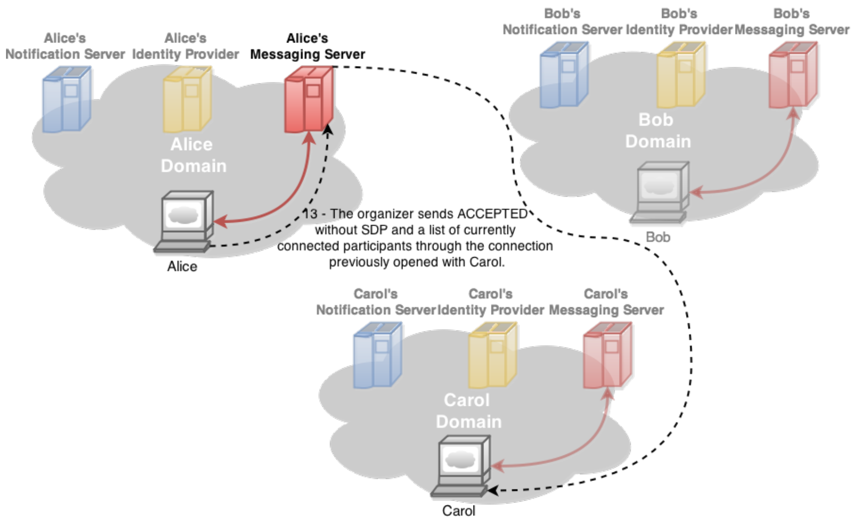

Alice's Notification Server
Alice's Identity Provider
Alice's Messaging Server
Alice Domain
Alice
13 - The organizer sends ACCEPTED without SDP and a list of currently connected participants through the connection previously opened with Carol.
Bob's Notification Server
Bob's Identity Provider
Bob's Messaging Server
Bob Domain
Bob
Carol's Notification Server
Carol's Identity Provider
Carol's Messaging Server
Carol Domain
Carol

Figure B.13: Call establishment algorithm, step 6 - Multiparty with hosting

15. To make this connection to Bob, Carol will send an INVITE to Bob through Alice's MS.
16. Bob receives Carol's invitation and automatically accepts it.
17. Carol receives Bob's accepted and the connection between them gets established.

Note that in this algorithm it is important to maintain the processing of the incoming ACCEPTED message in the organizer (steps 7 and 12) and the update of the list of connected peers as an atomic operation, to ensure the interconnection between all peers.

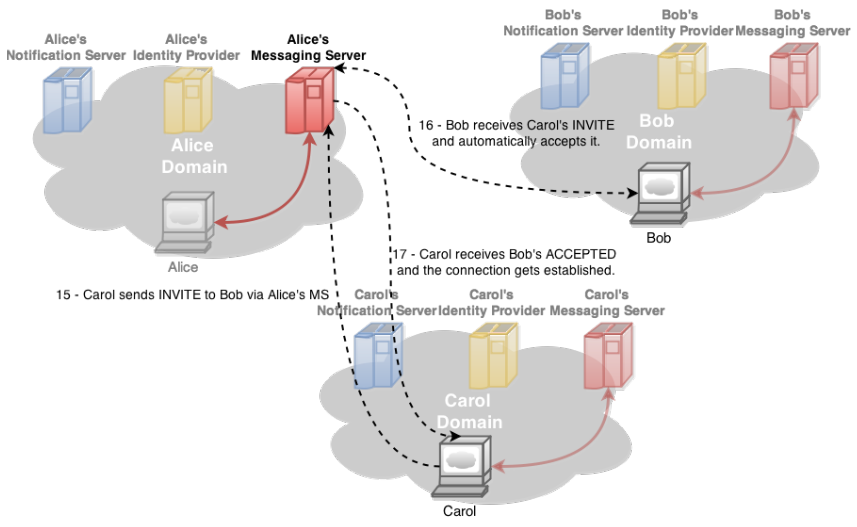

Alice's Notification Server
Alice's Identity Provider
Alice's Messaging Server
Alice Domain
Alice
Bob's Notification Server
Bob's Identity Provider
Bob's Messaging Server
Bob Domain
Bob
16 - Bob receives Carol's INVITE and automatically accepts it.
17 - Carol receives Bob's ACCEPTED and the connection gets established.
15 - Carol sends INVITE to Bob via Alice's MS
Carol's Notification Server
Carol's Identity Provider
Carol's Messaging Server
Carol Domain
Carol

Figure B.14: Call establishment algorithm, step 7 - Multiparty with hosting

Appendix C

Code Examples

C.1 Example app using the Conversation Layer

The following code example show how to create a simple bidirectionall audio/video communication app built on top of the highest abstraction layer of the interoperability

C.1.1 HTML Code

```html
<html>
<head>
    <title>Minimal Test Application</title>

    <!-- polyfill to switch-hit between Chrome and Firefox -->
    <script src="../../api/adapter.js"></script>
    <!-- Interoperability library classes -->
    <script src="../../api/Wonder.js"></script>


    <!-- application logic -->
    <script src="mini.js"></script>
</head>
<body>
    <h1>Minimal Test Application</h1>
    <div id="login">
        <input type="text" id="loginText" value="">
        <button id="loginButton" onclick="login()">Login</button>
    </div>
    <div>
        <div id="call">
            <input type="text" id="callTo" value="">
            <button id="callButton" onclick="doCall()">Call</button>
            <button id="hangup" onclick="hangup()">Hangup</button>
        </div>
    </div>
    <div id="videoContainer">
        <video id="localVideo" width="320" height="240"
```

```
          autoplay="autoplay" muted></video>
         <video id="remoteVideo" width="320" height="240"
          autoplay="autoplay" muted></video>
     </div>
</body>
</html>
```

C.1.2 JavaScript Logic

```
var localVideo;
var remoteVideo;
var myIdentity;
var conversation;

/* Definition of the STUN and TURN servers that are used for the setup of the
    RTCPeerConnection */
var STUN = {url: "stun:stun.server.ip:port"};
var TURN = {
    url: "turn:turn.server.ip",
    username: "username",
    credential: "password"
};
var iceServers = {"iceServers": [STUN, TURN]};


/* Definition of the constraints for the initial creation of the RTCPeerConnection, in
    this example the conversation is requested with audio/video in both directions */
var constraints = [{
    constraints: "",
    type: ResourceType.AUDIO_VIDEO,
    direction: "in_out"
}];


// informational callbacks from WebRTC engine
onCreateSessionDescriptionError = function(){ console.log("Error on Session description
    creation")};
onSetSessionDescriptionError = function(){console.log("Error on Session description
    assignment")};
onSetSessionDescriptionSuccess = function(){console.log("Session description success")};


/* This method performs the main initialization logic.
- It uses the IdP to create an Identity object from the entered URI
- It resolves the Identity, i.e. downloads the corresponding messagingStub for the users
    domain.
- It establishes the connection between the stub and the domains Messaging Server.
*/
function login() {
    var myRtcIdentity = document.getElementById('loginText').value;
    localVideo = document.getElementById('localVideo');
    remoteVideo = document.getElementById('remoteVideo');
    // bind main event listener listener
    var listener = this.onMessage.bind(this);
    // create own Identity
    Idp.getInstance().createIdentity(myRtcIdentity, function(identity) {
        // keep reference for later use
        myIdentity = identity;
        // download and instantiate (own) MessagingStub
        myIdentity.resolve(function(stub) {
            stub.addListener(listener);
            // connect own Stub to own domain
            stub.connect(myRtcIdentity, "", function() {
                console.log("own stub connected");
            });
        });
    });
}



/* This method performs all required actions to establish the communication with
the user(s), represented by the entered URI(s). This includes:
- Requesting access to local media sources (camera, microphone)
- Resolving of the target URI(s) and downloading of the corresponding
messagingStub(s)
- Connection of the stub(s) with the target domains
- Sending of the invitation message to the target users
- Handling of response and establishment of the RTCPeerConnection
*/
function doCall() {
    var peers = document.getElementById('callTo').value.split(";");
    conversation = new Conversation(myIdentity, this.onRTCEvt.bind(this),
    this.onMessage.bind(this), iceServers);
    var invitation = new Object();
    invitation.peers = peers;
    conversation.open(peers, constraints, invitation);
}


/* This method is the callback for incoming signalling messages. In this minimal
example, it just handles incoming Invitations and Bye messages.
On incoming invitations, a confirmation dialog is displayed with the options to
accept or reject the call. The Bye handling just performs some cleanup actions. */
function onMessage(message) {
    switch (message.type) {
        case MessageType.BYE:
            localVideo.src = '';
            remoteVideo.src = '';
            conversation = null;
            break;
        case MessageType.INVITATION:
            var accept = confirm("Incoming call from: " +
            message.from.rtcIdentity + " Accept?");
            if (accept == true) {
            // Create new conversation object
            conversation = new Conversation(myIdentity,
            this.onRTCEvt.bind(this),
            this.onMessage.bind(this), iceServers,
            constraints);
            conversation.acceptInvitation(message);
            }
            else
            conversation.reject(message);
            break;
        default:
            break;
    }
};


/* This method is the callback for RTC Events. These events are triggered by the
WebRTC engine in the browser as result of the ICE negotiations between the peers.
The main events to handle are the "onaddstream", which indicates that a remote
stream was added to the RTCPeerConnection and the "onaddlocalstream" which is the
counterpart for locally added streams.
The implemented actions just assign the streams to the corresponding video-tags of
the html page.*/
function onRTCEvt(event, evt) {
    switch (event) {
        case 'onaddstream':
            attachMediaStream(remoteVideo, evt.stream);
            break;
        case 'onaddlocalstream':
            attachMediaStream(localVideo, evt.stream);
            break;
        default:
            break;
    }
};


/* This method ends the established communication and performs some cleanup. Main
instruction is "conversation.bye()" which sends a BYE message to the peer and takes
care of the RTCPeerConnection and local media cleanup. */
function hangup() {
    localVideo.src = '';
    remoteVideo.src = '';
    conversation.bye();
    conversation = null;
}
```

C.2 Example app using the Core Layer

In order to illustrate the main coding concepts when using the Core layer, the following example code shows how the well-known Web Real Time Communication (WebRTC) reference application on [34] could be adapted to use the messaging layer developed in this thesis. For the sake of readability not the full sources are shown but instead some code snippets are provided with a short explanation and their proposed allocation in the original "apprtc" code. To get the full original sources of the apprtc reference app, visit [34] and view the page sources.

```
/* Some variables in the global scope to keep some states.
*/
var myIdentity;
var peerIdentity;
var initiator;
var contextId;
var invitationMessage;


/* The Resource constrains to be used for the initial call establishment. */
var constraints = [{
    constraints: "",
    type: ResourceType.AUDIO_VIDEO,
    direction: "in_out"
}];


/* The initialization of the interoperability stack. This includes the creation of the
    own
Identity as well as the download and connection of the MessagingStub.
This snippet mainly replaces the openChannel() method in the original code. */
function initFramework() {
    var myRtcIdentity = "user@domain.com";
    // bind main event listener listener
    var listener = this.onSignallingMessage.bind(this);
    // create own Identity
    Idp.getInstance().createIdentity(myRtcIdentity, function(identity) {
        // keep reference for later use
        myIdentity = identity;
        // download and instantiate (own) MessagingStub
        myIdentity.resolve(function(stub) {
            stub.addListener(listener);
            // connect own Stub to own domain
            stub.connect(myRtcIdentity, "", function() {
                console.log("own stub connected");
            });
        });
    });
}


/* When an outgoing call is being initiated, the signalling Message of type INVITATION
must be sent.
This must happen in the success-callback of the pc.createOffer() method of the
original code. */
function doCall(){
    // PSEUDO-CODE!
    peerConnection.createOffer( function(localDescription) {
        peerConnection.setLocalDescription(localDescription);
        sendInvitation(callee, localDescription);
    });
}


/* When an incoming call is being answered, the Message of type ACCEPTED
must be sent.
This must happen in the success-callback of the pc.createOffer() method of the
original code. */
function doAnswer(){
    // PSEUDO-CODE!
    peerConnection.createAnswer( function(localDescription) {
        peerConnection.setLocalDescription(localDescription);
        sendInvitationAccepted(invitationMessage, localDescription);
    });
}


/* This method creates and sends an INVITATION message.
See doCall() for description of correct place in the original code. */
function sendInvitation(toUri, localDescription) {
    this.initiator = true;
    var that = this;
    Idp.getInstance().createIdentity(toUri, function(toIdentity) {
        that.peerIdentity = toIdentity;
        toIdentity.resolve(function(peerStub) {
            that.contextId = Math.floor((Math.random() * 100000) + 1);
            var invitationMessage =
            MessageFactory.createInvitationMessage(that.myIdentity,
            toIdentity, that.contextId, that.constraints);
            invitationMessage.body.connectionDescription= localDescription;
            peerStub.sendMessage(message);
        });
    });
}


/* This method creates and sends an ACCEPTED message.
See doAnswer() for description of correct place in the original code. */
function sendInvitationAccepted(invitationMessage, localDescription) {
    this.initiator = false;
    this.contextId = invitationMessage.contextId;
    this.peerIdentity = invitationMessage.from;
    var acceptedMessage = MessageFactory.createAnswerMessage(peerIdentity, "",
    invitationMessage.contextId, constraints);
    acceptedMessage.body.connectionDescription = localDescription;
    myIdentity.messagingStub.sendMessage(acceptedMessage);
}


/* This method creates and sends a CONNECTIVITY_CANDIDATE message.
This will happen in the "onIceCandidate" message of in the original code and would
replace the invocation of sendMessage(). */
```

```
function sendConnectivityCandidate(candidate) {
    var candidateMessage = MessageFactory.createCandidateMessage(myIdentity,
    peerIdentity, contextId, "label", "id", candidate);
    if (initiator)
        peerIdentity.messagingStub.sendMessage(candidateMessage);
    else
        myIdentity.messagingStub.sendMessage(candidateMessage);
}


/* The BYE method must be invoked inside the hangup() method of the original code.
*/
function sendBye() {
    var byeMessage = new Message(myIdentity, peerIdentity, "", MessageType.BYE,
    contextId);
    if (initiator)
        peerIdentity.messagingStub.sendMessage(byeMessage);
    else
        myIdentity.messagingStub.sendMessage(byeMessage);
}


/* This is the callback for the main incoming messages. This code mainly
replaces the one in the processSignallingMessage of the original example. The
message in the parameter is a message then.*/
function onSignallingMessage(message) {
    switch (message.type) {
        case MessageType.INVITATION:
            doAnswer();
            break;
        case MessageType.ACCEPTED:
            // setRemoteDescription()
            // perform GUI actions etc
            break;
        case MessageType.BYE:
            // cleanup WebRTC and GUI stuff
            break;
        case MessageType.CONNECTIVITY_CANDIDATE:
            // extract and create RTCIceCandidate from message
            peerConnection.addIceCandidate(candidate,
            onAddIceCandidateSuccess, onAddIceCandidateError);
            break;
        default:
            break;
    }
};
```

C.3 Messaging Stub Development

C.3.1 Connect() Method

```
MessagingStub_SimpleWebSocket.prototype.connect = function(ownRtcIdentity,
credentials, callbackFunction) {
    if (this.websocket) {
        console.log("Websocket connection already opened");
        callbackFunction();
        return;
    }
    this.websocket = new WebSocket("ws://host:port");
    this.websocket.onopen = function() {
        var message = new Object();
        message.type = "login";
        message.from = ownRtcIdentity;
        socket.send(JSON.stringify(message));
        callbackFunction();
    };
    this.websocket.onerror = function() {
        console.log("Websocket connection error");
    };
    this.websocket.onclose = function() {
        console.log("Websocket connection closed");
    };
    var that = this;
    this.websocket.onmessage = function(full_message) {
        /* Extract the signalling message from the domain specific protocol. Just
        takes the "body" part of the received message as content.*/
        var message = JSON.parse(full_message.data).body;
        /* Use the Idp to create Identity objects from the string
        representations in the from and to fields of the message. */
        Idp.getInstance().createIdentity(message.from, function(identity) {
            message.from = identity;
            Idp.getInstance().createIdentities(message.to,
            function(identityArr) {
            message.to = identityArr;
            // forward the message to main API message chain
            that.baseStub.deliverMessage(message);
        });
    });
};
```

C.3.2 SendMessage() Method

```
MessagingStub_SimpleWebSocket.prototype.sendMessage = function (message) {
    /* Here, the very simple "translation"/wrapping into the domain specific
    message format happens. The message is just put to the body-field of
    the surrounding full_message.*/
    var full_message = new Object();
    full_message.type = "message";
    full_message.body = message;

    /* From and To Identities are changed into strings containing rtcIdentities */
    message.from = message.from.rtcIdentity;
    if (message.to instanceof Array) {
        message.to.every(function (element, index, array) {
            array[index] = element.rtcIdentity;
        });
        full_message.to = message.to[0];
    } else {
        message.to = new Array(message.to.rtcIdentity);
        full_message.to = message.to[0];
    }
    // send via the websocket that was established in connect()
    this.websocket.send(JSON.stringify(full_message));
};
```

C.3.3 Disconnect() Method

```
MessagingStub_SimpleWebSocket.prototype.disconnect = function() {
    this.websocket.close();
    this.websocket = null;
};
```