Freie Universität Berlin

Institute for Computer Science
Takustr. 9
14195 Berlin

Bachelor Thesis

Enabling Browser-Based Real-Time Communication for Future Internet Services

WebRTC and OAuth Capabilities for FOKUS Broker

Georg Graf

Matriculation Number: 4282989

18.06.2012

Supervised by
Prof. Dr.-Ing. Jochen Schiller
FU Berlin

Prof. Dr. Thomas Magedanz
TU Berlin

Assistant Supervisor
Alexander Blotny
Daniel Gona
FhG FOKUS

FOKUS Institute
Kaiserin-Augusta-Allee 31
10589 Berlin

This Bachelor thesis originated in cooperation with the Fraunhofer Institute for Open Communication Systems (FOKUS).

First of all I would like to thank Prof. Dr. Thomas Magedanz at the Fraunhofer Institute FOKUS for giving me the opportunity to carry out state of the art research in the field of WebRTC and OAuth.

Special thanks to Alexander Blotny and Daniel Gona for their guidance throughout the work and research on this thesis, providing valuable input and criticism as well as approval at the right moments, thereby helping me also overcome some hurdles in the process.

Furthermore I would like to thank Dr. Niklas Blum, who gave me useful feedback on my drafts and Thomas Pietsch, who always was a competent contact on issues with IMS, SIP and other technologies I had to deal with.

Hereby I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Berlin, 18.06.2012

. . . .

Georg Graf

Abstract

In recent times the World Wide Web is highly driven by the emergence of new HTML5 features. The fast standardization process of these features makes the new generation of browsers highly independent of external third-party plugins and thus making them safer for users and easier for developers. The standards are developed by the Internet Engineering Task Force (IETF)[1], the World Wide Web Consortium (W3C)[2] and Web Hypertext Application Technology Working Group (WHATWG)[3]. One new set of standards, known as the Real Time Communication for the Web (WebRTC)[2] is currently in work and allows real peer-to-peer communication between browsers and access to the users camera. A related standard called WebSocket [4] enables real bidirectional connection from browser to server.

The FOKUS Broker [5] is a platform designed to enable the combination of Internet and telecommunication services with the ability to control the service access and execution through enforcement of policies and exposing the resulting services again via standardized APIs. It is based on Service Oriented Architecture (SOA)[6] principles. The FOKUS Broker uses as fundament for the telecommunication services the IP Multimedia Subsystem (IMS)[7] a Next Generation Network (NGN)[8] infrastructure.

The main objective of this thesis is to establish the fundamentals for the FOKUS Broker to use the real time communication capabilities of future browsers. Especially the usefulness of WebSocket's will be evaluated closely with regard to performance and complexity. In the end it should be possible to communicate with SIP [9] clients from the browser via WebSockets. The browser-to-browser video and audio communication is another milestone that should be achieved. Utilizing the telecommunication capabilities the FOKUS Broker provides and incorporating them into the WebRTC context is another goal. The newly acquired capabilities should be combinable with other third-party Internet services and preferably make use of their capabilities in an authorized manner, there the OAuth [10] Protocol can probably be beneficial.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

In the present time the Internet, especially the World Wide Web, is undergoing a rapid change towards specifying and implementing HTML5 features. Whereas W3C (World Wide Web Consortium)[2], the WHATWG (Web Hypertext Application Technology Working Group)[3] and the IETF [1] (Internet Engineering Task Force) are the main forces behind developing the standards, there are also several companies and foundations, like Google[11], Mozilla[12], Ericsson[13], Apple[14] and Opera[15], working with them to produce practical and high-quality specifications. They are highly invested in driving these standards forward by implementing their features, developing prototypes and generating demand in the ranks of Internet users through fascinating demos and hardware devices, relying on integration of some HTML5 standards, in order to make the web a more open and free environment (i.e. [16]). An even larger goal is to accomplish convergence between the web, the mobile web and the telecommunication world.

Part of the new HTML5 features, which has the potential to revolutionize [17, 18] the communications in the web, is the set of standards known as WebRTC (Web Real Time Communication) [2][11]. It enables and simplifies the real time communication inside the web-browser, allowing services ranging from simple streaming of data to audio- and video-conferencing as well as online games.

The development of WebRTC is right now divided in two areas, which are dependent on each other. On the one side is the browser development, where Google, with their WebRTC C++ library for browser developers is advancing rapidly with implementing the specifications the rtcweb group [1] of IETF is proposing. On the other side there are the JavaScript APIs, which are developed by the W3C and WHATWG and implemented in experimental builds of browsers from Google (Chromium Browser[19]) and Ericsson (Epiphany, GTKLauncher based on a modified WebKit library[20]). An overview of this can be seen in figure 1.1.

Another distinction can be made in the two paths the WebRTC implementation takes. For one thing there is the signaling path, which goes over a web server and is an upgraded HTTP connection – a WebSocket, a finalized standard [4] by the IETF and implemented in most modern browsers. The other is the media path – a peer-to-peer connection from one WebRTC client to another, which can be a browser or any other application that

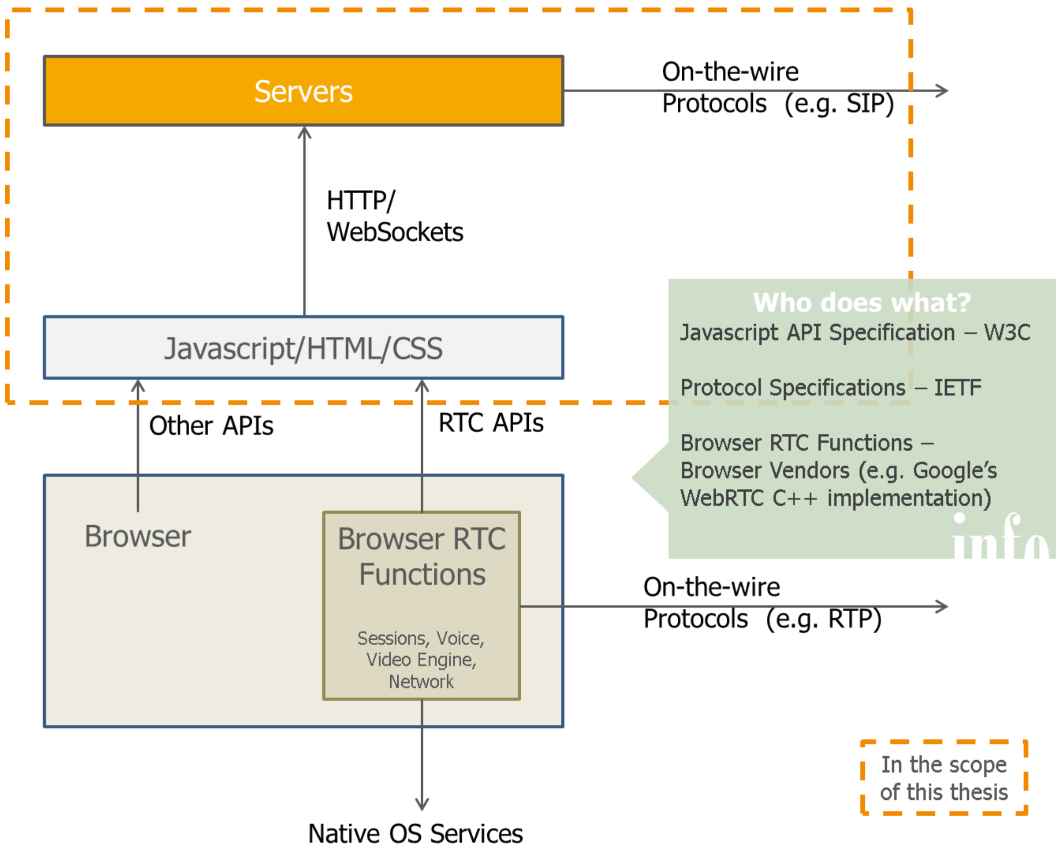

Servers
On-the-wire Protocols (e.g. SIP)
HTTP/ WebSockets
Javascript/HTML/CSS
Who does what?
Javascript API Specification – W3C
Protocol Specifications – IETF
Browser RTC Functions – Browser Vendors (e.g. Google's WebRTC C++ implementation)
info
Other APIs
RTC APIs
Browser
Browser RTC Functions
Sessions, Voice, Video Engine, Network
On-the-wire Protocols (e.g. RTP)
In the scope of this thesis
Native OS Services

Figure 1.1: Overview of WebRTC

implements the rtcweb standard of IETF. This can also be seen in the low path of figure 1.1 and even better in figure 1.3.

The goals of the WebRTC HTML standards are on the one hand – revising the browser-to-server communications [18]; on the other hand – innovating the direct browser-to-browser communication [11]. The old approach to communicate directly between server and browser without reloading a page had to be simulated through JavaScript with different techniques (like long-polling or other Comet [21] methods). There are several problems coming along with this, e.g. a high latency, an unnecessary large overhead of data and the complexity in the implementation of the server-side and client-side code[18][17]. It is even worse in the case of browser-to-browser communication; there is no real plugin-free way of establishing a direct connection between two browsers on the Internet right now. Some Flash implementations and several plugins like “Google Talk” exist, which can solve some use cases that could be imaginable with a direct point-to-point connection, but this is all non-standardized, platform dependent and requires workarounds in the implementation to make it possible.

This thesis will mostly concentrate on the solution for the server-to-browser communication of the WebRTC topic, which is the WebSocket. The WebSocket is supposed to solve all the three mentioned problems of today's implementations of bi-directional communication [18][17]. As the Fraunhofer FOKUS (Fraunhofer Institute for Open Communication Systems) NGNI CC (Next Generation Network Infrastructures Competence Centre) is aspiring to be in a leading role regarding the expertise in the future Internet development [22], it is interested in utilizing this technology and its potentials. This thesis will explore how it could be beneficial in the context of the FOKUS Broker [5], which is a platform designed to enable the combination of Internet and Telecommunication components, by providing different Service Interfaces for third party developers.

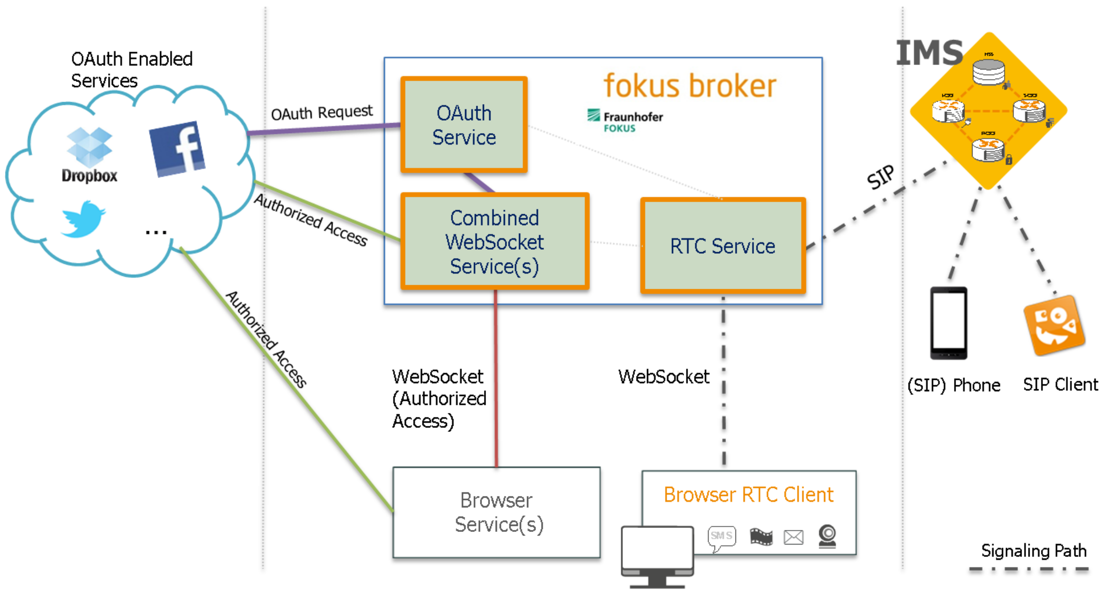


Figure 1.2: Full Scope

1.2 Goals and Objectives

The primary objectives of this thesis are to explore and document the possibilities, which WebRTC and WebSocket's [4] in particular are opening up for the FOKUS Broker (hereafter referred to as Broker). This involves implementing simple use cases, measuring their performance, evaluating security concerns and utilizing fallback mechanisms for browsers which do not support them.

A further goal is to enable communication with the underlying IMS (IP Multimedia Subsystem)[7] network from the browser (see figure 1.3) to make real time communication (like instant messaging and later audio/video call) possible. This shall be combined with the ability to interact with other Third-Party online services in an authorized manner to enable a wider variety of applications.

Implementing a Broker Service that is used to acquire, store and provide Authorization Tokens to other services inside the Broker is the goal here.

The combination of this new WebRTC functionalities with existing Broker services and other Third Party services through OAuth should be demonstrated in the end as a combined use-case.

The full scope can be seen in figure 1.2. The area between the grey dotted lines indicates the parts that are being implemented in the course of this thesis (except the FOKUS Broker).

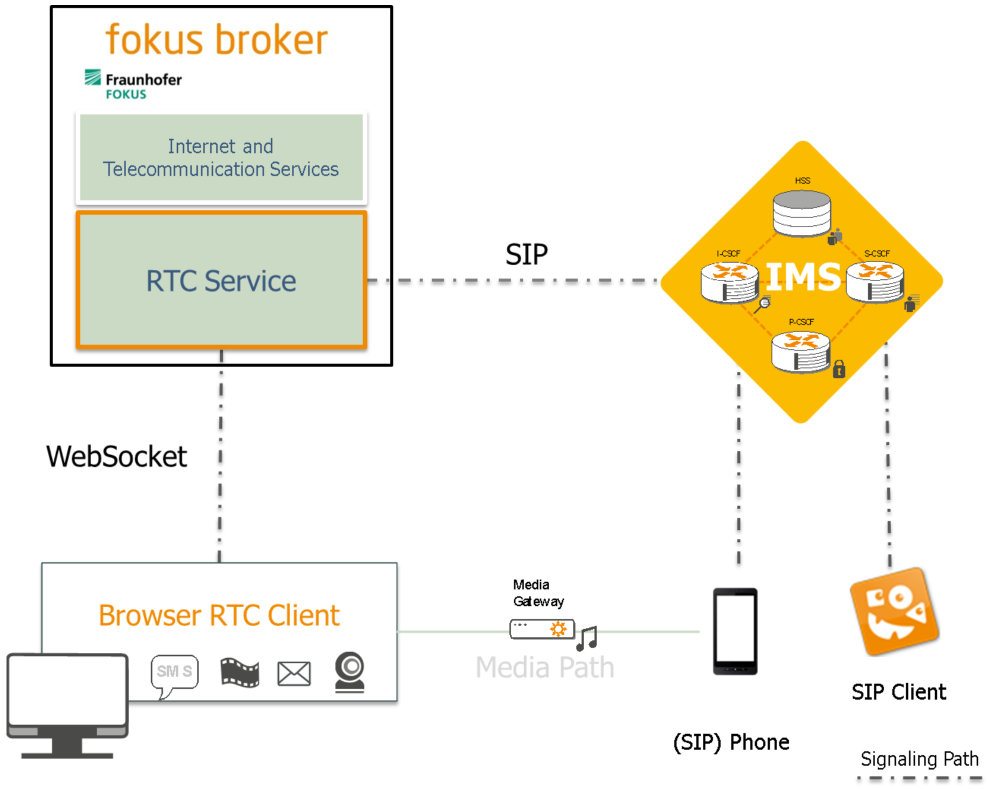


Figure 1.3: RTC Service

1.3 Out of scope

The topic WebRTC is rather large, so it is necessary to define what is out of scope for this thesis. The MediaStream and PeerConnection, which establish a connection between browsers and are responsible for the Real Time Stream, which is sent to another WebRTC client are not part of this thesis and will be covered briefly (see Media Path in figure 1.3). Although this so called Media Path is an important part, the focus in this work lies on the Signaling Path and therefore on WebSocket. This work will be a fundamental milestone in the complete implementation of all WebRTC capabilities for the FOKUS Broker. The decision to leave this part out was based on the lack of browser implementations in this area.

The peer-to-peer connection between a Browser and a normal SIP Client or a SIP Phone is out of scope, because a gateway would be needed or an adaptation of the SIP Client/-Phone implementation, but a use-case utilizing a direct connection between two browsers might be possible in some use-case implementations depending on how quick the other objectives are met and on the implementation status of the MediaStream and PeerConnection API in a Browser.

1.4 Artifacts

Here is a list of expected artifacts that should come out from this thesis:

- RTC Service for the Broker with a WebSocket Interface (with documentation of usage, design and the security aspects)
- RTC Client in the Browser (Documentation of the Implementation)
- OAuth Service for the Broker as a *.jar and/or a *.war file (with documentation and design documents)
- Some example Broker services utilizing WebSockets and the OAuth Service (and the SIP Service) and documentation, also an evaluation of their performance and code complexity, as well as security aspects

1.5 Outline

This thesis is separated into 7 chapters.

In **Chapter 2** all the technology and software that is involved in the development process will be described. Also the current development status of the specifications and implementations of various standards in the field of WebRTC and OAuth is the topic here.

Chapter 3 deals with the requirements of the software components that are implemented in the course of this thesis. How should they work and how should they be used.

Chapter 4 proposes a design for the services and discusses why this is considered the best solution.

Chapter 5 describes the implementation of the desired artifacts.

In **Chapter 6** the implementations will be evaluated for performance and usability. Use-cases will be implemented to show how the components are used, and to put them in a less abstract context.

At last in **Chapter 7** the work done in this thesis will be reviewed and problems that occurred during the implementation and design will be mentioned. Also an outlook on future work will be given.

2 State of the Art

2.1 WebRTC

2.1.1 What is WebRTC?

WebRTC (Web Real Time Communications) is a set of specifications that are currently being developed by the IETF[1], W3C [2]and WHATWG [3] enabling real time peer to peer communications in the web browser, (whereas IETF develops the protocol specifications and therefore uses the term RTCWEB [1]). One of the most desired use cases is the ability for audio and video conferences inside the browser without any plugin, which will be enabled by WebRTC [2]. Other applications inside the browser depending on a real time communication like games, peer-to-peer networks, financial monitoring or device monitoring will also benefit from it.

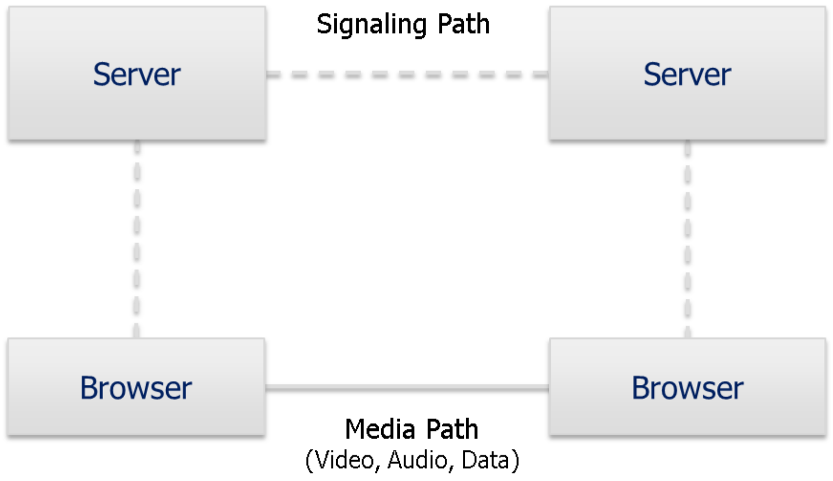


Figure 2.1: Basic Architecture()

The basic idea is to enable a Peer-to-Peer connection between browsers. To make this possible a signaling path between the two browsers needs to be established. This has to happen over a server (or multiple servers). The protocol of this signaling channel is proprietary and how it is realized is up to the programmer, the only thing it has to guarantee is that the signaling messages arrive in the right order. It iSs a good and convenient idea to use WebSockets for the signaling channel in the browser, but one can also use XMLHttpRequests(XHR) and long polling or other techniques. The distinction between the two paths can be seen in figure 2.1.

2.1.2 Signaling

The signaling in WebRTC is currently a heavily discussed topic. The first utilized protocol was ROAP (RTCWeb Offer/Answer protocol)[23]. ROAP defines a strict state machine how signaling occurs. It takes most of the work of session initiation, creating offers and answers out of the web developers' hand. Therefore it is really easy to implement RTC applications inside the browser with it. The downside is the little control the developer has to customize the session initiation, choosing codecs or making configurations. Inside ROAP the Session Description Protocol (SDP) [24] is used for negotiating codecs.

The IETF decided this signaling process was too strict lately and introduced [25] a new approach called JSEP - the JavaScript Session Establishment Protocol [26]. In JSEP the state machine for signaling is not so strict anymore and the ICE Agent and Signaling agent got separated.

ICE stands for Interactive Connectivity Establishment; it is a protocol for NAT (Network Address Translator) Traversal for Offer/Answer protocol and defined in RFC5245[27].

What an ICE Agent does in context of WebRTC in the browser, is the mentioned NAT Traversal by using a STUN (Session Traversal Utilities for NAT) [28] or TURN (Traversal Using Relay NAT)[29] Server, it collects candidates (addresses) for the partner to whom a PeerConnection gets established. Once the connection is established it processes the SDP messages passed to it, it can also create SDP offers and answers.

In JSEP the important thing is to pass the incoming SDP messages to the ICE Agent, how the signaling is done and wrapped is up to the developer. There are JavaScript libraries that implement ROAP, also it is possible to wrap it into SIP messages (but this still has to go over a Server to convert them to real SIP Messages) or XMPP. Now that the ICE Agent is separated from the signaling state machinethe signaling can start earlier, because the application does not have to wait until all candidates got collected. Also it is now possible to pass Media Hints to the ICE Agent when creating an offer, so certain codecs can be preferred. [25]

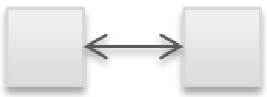
To conclude it can be stated that JSEP is not a real protocol but more an approach how signaling is handled. The main advantage is the decoupling of transport and actual signaling. It is very flexible and leaves most things to the developer. Less experienced developers can still use JavaScript libraries to simplify the process.

2.1.3 Media Path

In the Media Path of WebRTC the Real Time Protocol (RTP) is used. According to the specification draft [30] the following topologies are considered:

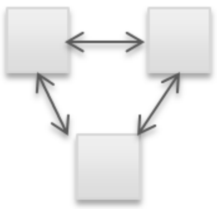
Point-to-point:

This is the standard communication use case – a single point-to-point connection between two users with their clients implementing WebRTC.



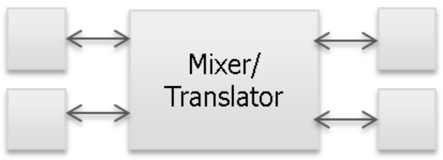
Multi-Unicast:

In this topology multiple PeerConnections are established between a manageable number of users. The downside is that with increasing amount of users the bandwidth requirements rise.



RTP Mixer with only Unicast paths:

An RTP Mixer is a centralized point acting as a RTC client and can be used for conferences to control the audio or video that is sent and optimize used bandwidth by mixing together multiple streams into one stream for example.



Translator (Relay) with only Unicast paths:

A translator is a central unit, which simply forwards the incoming streams to the other connected clients.

RTP Translator towards legacy systems:

This topology is for interoperability with legacy end points (e.g. phone, fax). Between the clients is a central unit that translates the RTP stream into the appropriate signal for the other end-point. From the perspective of the WebRTC client, the translator looks like another WebRTC client.

The required codecs used in the RTP stream are chosen to enable interoperability with legacy systems. The mandatory audio codecs are PCMA/PCMU, which is a standard telephony codec, Telephone Event, which handles audio control signals, and Opus. The video codecs are still under heavy discussion only their parameters are agreed upon; the actual codecs are not decided yet. The discussion is about whether license free or commercial codecs should be used.

To understand WebRTC you have to distinguish between the browser implementation and the JavaScript APIs. The JavaScript APIs will be used by web developers and are specified by the W3C. In the taskforce designated to write this standard are members of Ericsson [13], Voxeo [31], Cisco [32] and Mozilla [12]. The browser implementation on the other hand depends on the IETF specification. It defines the protocols, security mechanisms and the general architecture. Working on it are also staff of Google, Cisco, Mozilla, Voxeo and Ericsson. Both specifications are highly interdependent and change frequently as the specification evolves, so the following description is just a snapshot of the current APIs.

2.1.4 JavaScript APIs

As previously mentioned WebRTC is just the collective term for multiple specifications namely the PeerConnection API, Stream API, getUserMedia and other related JavaScript APIs which will be explained in more detail in the following paragraphs. The JavaScript API is fairly simple and designed to be used by web developers, therefore most of the signaling and negotiating of codecs and protocols is done by the browser itself, although this is about to change with the introduction of JSEP as new signaling paradigm. (This section is based on the WebRTC API specifications [33] and own implementation experience.)

Stream API

The MediaStream object handles streams of media (audio, video). Each MediaStream has MediaStreamTrack which can be sent to a remote user over the PeerConnection, which will be explained later (2.1.4). Each MediaStreamTrack can consist of several channels, for example for stereo or 5.1 sound. With a MediaStream it is possible to mute individual MediaStreamTracks to control, which data is sent to a remote peer over a PeerConnection.

A MediaStream can be created by using the getUserMedia() call; in this case it will take the local user camera and microphone as input data. The getUserMedia functionality is specified in another W3C draft [34].

For the MediaStream to show up on a webpage an URL has to be created via createObjectURL(MediaStream stream) and then put as source into a video or audio html tag.

The MediaStream and MediaStreamTrack have various other functions, subtypes and attributes which will not be discussed here in further detail, because they are out of scope for this thesis, for more information kindly reference to the specification [2].

There will be also an object for recording video or audio called MediaStreamRecorder.

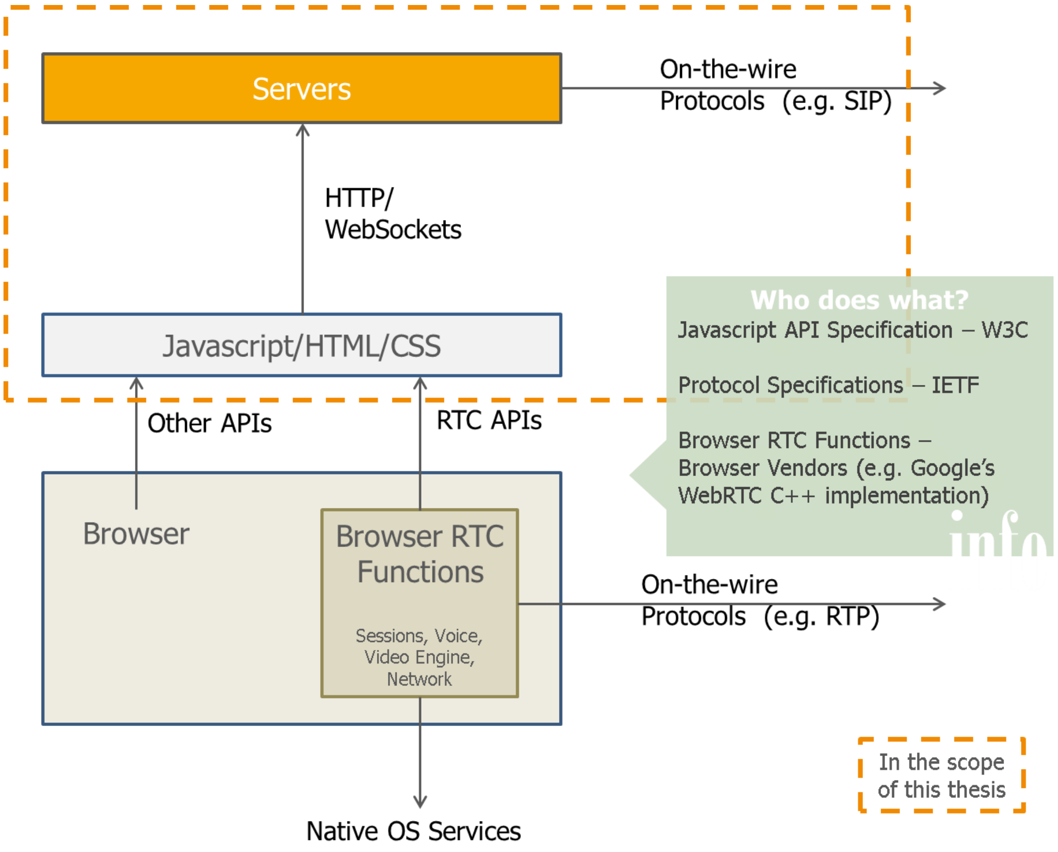

Servers
On-the-wire Protocols (e.g. SIP)
HTTP/ WebSockets
Javascript/HTML/CSS
Other APIs
RTC APIs
Who does what?
Javascript API Specification – W3C
Protocol Specifications – IETF
Browser RTC Functions – Browser Vendors (e.g. Google's WebRTC C++ implementation)
info
Browser
Browser RTC Functions
Sessions, Voice, Video Engine, Network
On-the-wire Protocols (e.g. RTP)
In the scope of this thesis
Native OS Services

Figure 2.2: Overview of WebRTC

PeerConnection API

In the heart of the WebRTC specifications lies the PeerConnection [35]. The PeerConnection allows a direct browser-to-browser communication. Before a direct connection can be established the PeerConnection needs a channel, which handles all the signaling and negotiation.

One part of the negotiation is the IP address of both users. Since users can be hidden behind a NAT (Network Address Translation) or a firewall there is a build-in mechanism to find out the real address of the user. When a PeerConnection Object is created the first argument is an address to a TURN (Traversal Using Relays around NAT) or a STUN (Session Traversal Utilities for NAT) server.

The second argument is the method, which is called when this PeerConnection wants to signal something to the other user. Arriving signaling messages get processed by calling the processSignalingMessage(message) method of the PeerConnection.

Once the PeerConnection is established there is the possibility to add MediaStreams. When a MediaStream is added, the PeerConnection will negotiate codecs over the signaling channel. After the negotiation an event is triggered on the other side of the connection, which indicates that a MediaStream was added and the Web Developer can decide, what to do with it.

2.1.5 Implementation in browsers

The WebSocket (section 2.1.6) Implementation is already a finalized RFC and is already implemented in most modern browsers like Chrome, Safari, Firefox and Opera.

The WebRTC standards on the other hand are not implemented in many browsers right now. The first implementation was a modified WebKit build released by Ericsson [36], which demonstrated WebRTC functionalities and had MediaStream and PeerConnection Implementations.

Google Chrome has implementations [38], which change almost daily in there Chrome Canary builds, as well as in the Chrome Dev Build on Linux systems. The Chrome implementation uses the WebRTC C++ API for browser developers released by Google. The C++ implementation already provides a voice and video engine, handles transport protocols, NAT Traversal and session management. The architecture of this C++ implementation is depicted in figure 2.3.

Right now the features are very experimental and usually only work if the browsers are the same versions (Chrome).

Opera has just implemented the getUserMedia in their Opera Mobile 12 browser [39] and can now display local video from the webcam in their browser. But the PeerConnection is not yet implemented.

Mozilla has now (03.04.2012) a WebRTC demo in a new build of FireFox, which simulates getUserMedia and PeerConnection, but is not fully standard compliant at the moment.

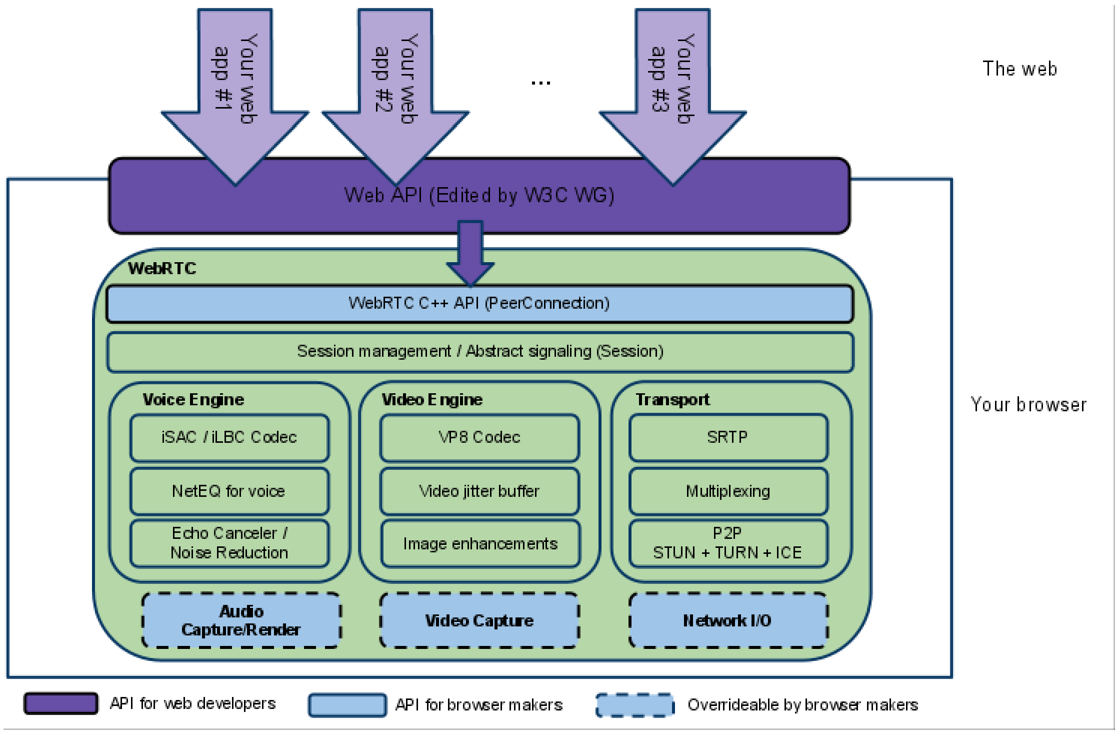


Figure 2.3: Browser Architecture [37]

2.1.6 Related APIs

WebSocket API

The new WebSocket standard [4] enables new possibilities for communication between a server and a browser. It is a bidirectional communication channel, so both entities can send data through it at any time. It is designed to behave like a TCP Socket, thus the reference in the name, and in its core it is TCP Connection, which is an upgraded HTTP Connection on the Application layer. This behavior was previously only emulated with different techniques. The easier part was to send data to the server; this could be done through POST, GET (in the parameters) or AJAX if it had to be done asynchronously. To push data from the server several possibilities were thought up e.g. HTTP Streaming, long polling or hidden iframes.

The WebSocket protocol starts with an HTTP, which gets upgraded to a WebSocket in the course of an WebSocket handshake. An example of such a request and a response can be seen in listing 2.1 and 2.2.

Listing 2.1: Browser WebSocket Request

```
GET ws://echo.websocket.org/?encoding=text HTTP/1.1
Origin: http://websocket.org
Cookie: __utma=99as
Connection: Upgrade
Host: echo.websocket.org
Sec-WebSocket-Key: uRovscZjNol/umbTt5uKmw==
Upgrade: websocket
Sec-WebSocket-Version: 13
```

Listing 2.2: Server WebSocket Response

```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Fri, 10 Feb 2012 17:38:18 GMT
Connection: Upgrade
Server: Kaazing Gateway
Upgrade: WebSocket
Access-Control-Allow-Origin: http://websocket.org
Access-Control-Allow-Credentials: true
Sec-WebSocket-Accept: rLHCkw/SKsO9GAH/ZSFhBATDKrU=
Access-Control-Allow-Headers: content-type
```

Once upgraded, the connection is framed into the WebSocket protocol, which has little overhead, just two bytes. That is only a fraction of the overhead an HTTP request/response headers produces.

The Kaazing [40] company has put some effort into evaluating the performance and listing the upsides of WebSockets. The insights are shared in an article [18], where they compare the performance of WebSockets with the Comet [21] approaches (Polling, Long-Polling and Streaming). The main downsides they mention and evaluate of these pre-websocket approaches are:

- **overhead**:
  The HTTP requests can have hundreds of bytes of overhead compared to just 2 bytes in a WebSocket framing. If projected on a high frequency of data exchange it makes an enourmous difference in network traffic especially in cases were only small amounts of data needs to be sent back and forth frequently, like some control data in games for example.

- **complexity**:
  The Complexity arises from the fact that often two connections for up- and downstream have to be managed to simulate a full-duplex connection between server and browser. This overhead in code length and implementation complexity makes it more difficult to program fully dynamic web services. It also produces unnecessary network traffic and increases CPU load.

- **latency**:
  Another aspect is the latency that is very important for real time applications.

These are also the areas where the new WebSocket technology has its strong points.

The issue with the caching proxies that cache HTTP responses could be considered a downside or an upside for WebSockets. The upside of caching is that the traffic to the server is reduced and frequently requested resources are returned faster. The downside is that these resources can be outdated.

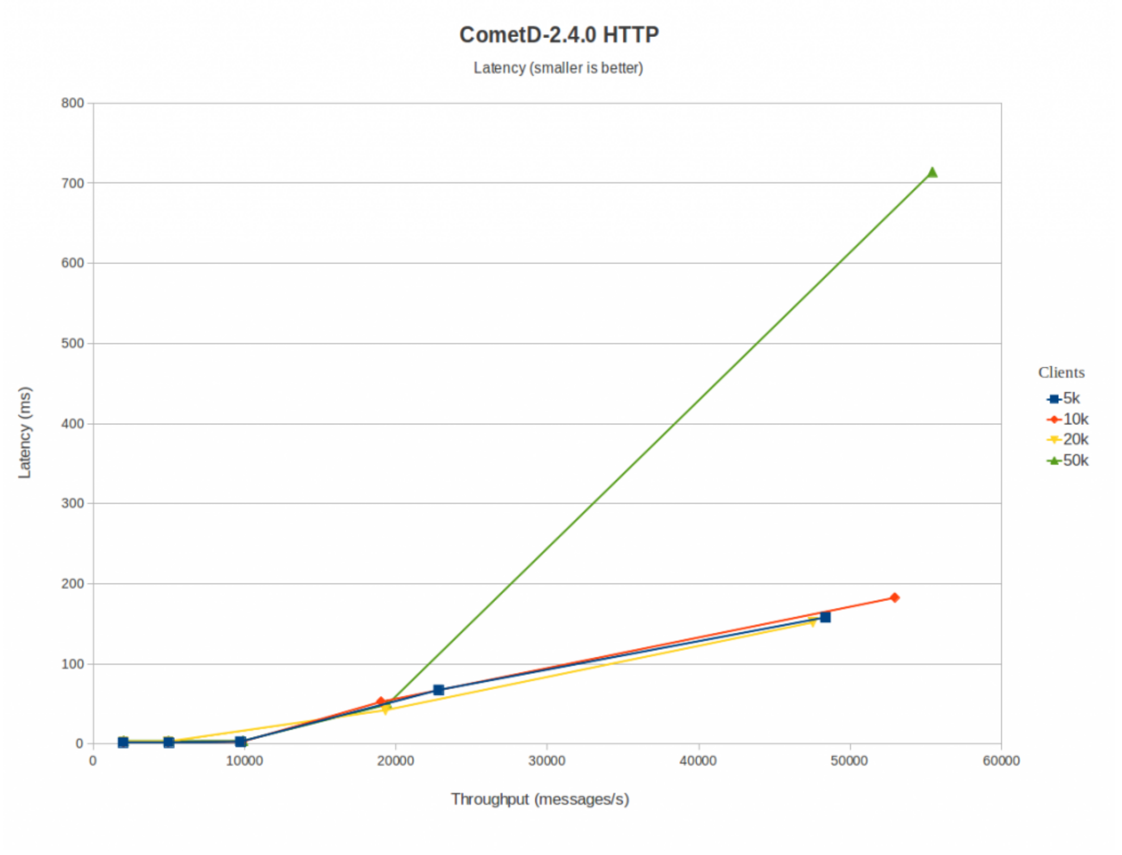


Figure 2.4: Performance test with CometD HTTP(graphic taken from [17])

In a blog of Jetty developers the latency was benchmarked and some load and performance testing was executed [17] and as a result some convincing data was gathered, underlining the advantages of WebSockets in comparison their CometD long-polling HTTP approach. What they measured was the latency depending on the throughput of messages per second (mps) and number of clients. For the load tests they used Amazon EC2 service. The results of their tests were quite impressive (see figure 2.4 and 2.5. The CometD implementation could handle client numbers up to 20k (20000) with linear increase of the latency along with increasing throughput with up to 50k mps. At

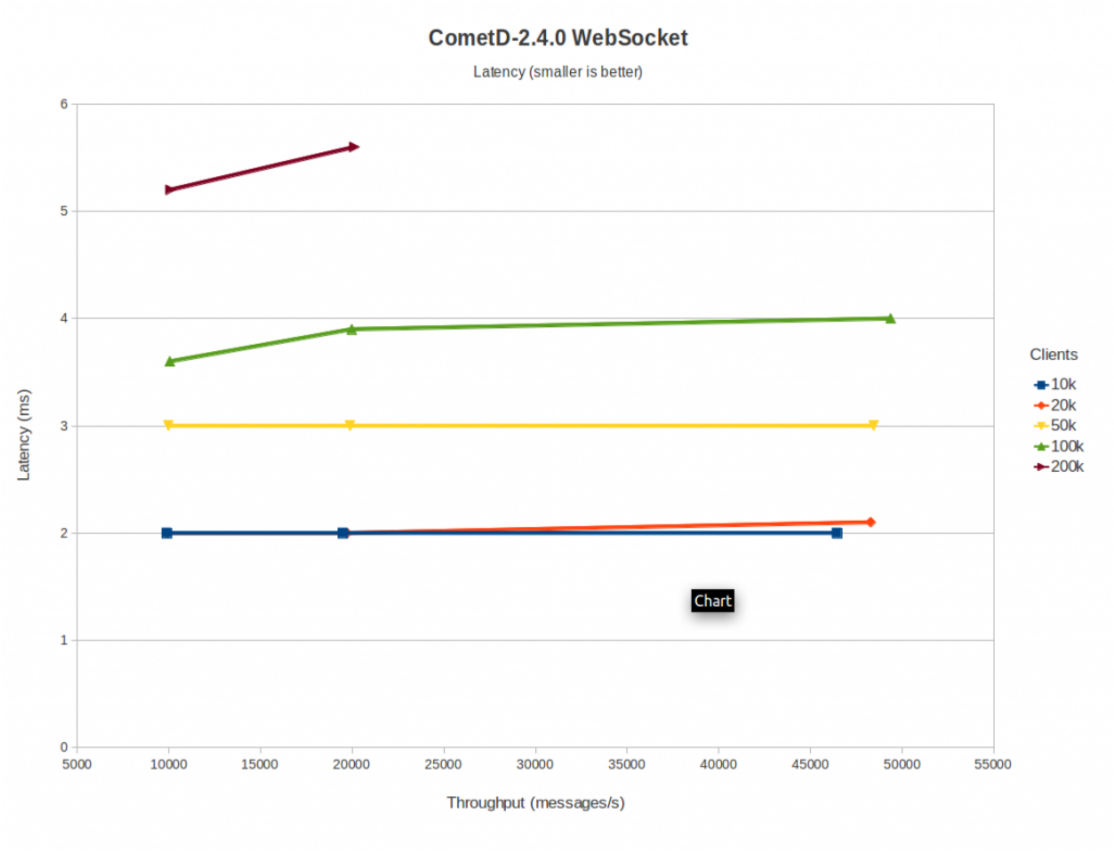


Figure 2.5: Performance test with CometD WebSocket (graphic taken from [17])

50k mps the latency was slightly under the 200ms mark. Under 10mps the latency is around 2-4ms. With websockets the latency stays constant around 2-5ms for increasing throughput of messages (up to 50k mps) and a much higher number of clients is possible (up to 200k).

The memory occupation of the HTTP approach is also significantly higher. For 50k clients it is aroud 2.1 GB, whereas WebSockets only occupy around 1.2GB for the same number of clients and 3.2GB for 200k clients.

2.2 OAuth

In today's rising amount of Internet services and, as a consequence, Internet identities, there is a desire for combination and interaction of those services with each other as well as third-party web, desktop and mobile applications without exposing the user credentials to them. Therefore the open protocol standard OAuth [10] was developed by the IETF [41] with combined expertise from previous proprietary protocols like Google AuthSub, Yahoo BBAuth and FlickrAuth [42]. OAuth stands for open authorization. It is a protocol for delegating specific authorizations from provider services holding important resources to third-party applications, without ever having the need to expose the user credentials to them, since the actual authentication and authorization happens at the providers. The user has full control and overview over the authorizations he gives to any application. In the following sections the OAuth principles will be explained in further detail, most of it based on the explanations [43] by one of the creators of OAuth himself Eran Hammer-Lahav and the RFC [41] and drafts [44].

2.2.1 Advantages

There are a lot of advantages when using the OAuth approach. The consumer benefits from the convenience to login with identities that he already has on sites like Twitter, Google or Facebook, relieving him from going through a process of account creation. Since the OAuth protocol evolved from the OpenID [45] community this functionality is a logical consequence. The OAuth protocol does not render OpenID useless; both standards can coexist and cooperate, as seen for example with Google. Google is an OpenID Provider, which lets users authenticate their OpenID with an OAuth request [46].

The core feature of OAuth is the authorization. Users can allow applications to make authorized requests to services for getting protected resources or invoke actions on behalf of the user. The main advantage of OAuth is that the user does not have to share his user credentials with the application requesting this authorization. He also can see which type of access or information the application is trying to get and if he wants this to happen. He can also revoke the granted rights at any time, without changing the password and thereby compromising the functionality of other interoperating services.

For application and web developers it brings the advantages of being a standard way to access protected APIs and user information, so that a developer only has to learn the basic principle of it once and can use it on numerous APIs all over the web. Also the barrier for the user to authorize the service is reduced, now that the user does not have to use his credentials and has full insight about what the service has access to.

At last, the Service Providers benefit from the standard, because it enables an easy and standard way to make their services interoperable, thus increasing value of their

Consumer	App and Web Developer	Service Provider
Convenient login/register	Standard way to access protected resources/APIs	Enhance interoperability
No sharing of user credentials	Learn once, use everywhere	Secure APIs
Insight into granted authorizations	Enhance user experience	Control/restrict access to API
Revoking authorizations of individual applications at any time (without breaking other connected applications)	Lower entry barrier	
Issue temporary authorizations		

Table 2.1: Advantages of OAuth

service, by expanding the potential number developers and as a result reaching more users. It is also a way to secure and restrict the access to the API. You can implement methods to enhance access for certain clients, so paying applications can get faster or more frequent access to the API for example.

A summary of the advantages is shown in table 2.1.

2.2.2 Basic principles

Roles

There are several roles defined in the OAuth process that should be first described. A rough overview is depicted in figure 2.6.

The first role is the Server. It is the OAuth Provider with the resources protected from unauthorized access. In OAuth 2.0 this role further is split into Resource Server and Authorization server. Whereas the Resource Server is the Server actually holding the protected resources, that can be only accessed with an authorization previously acquired from the Authorization Server.

The Client is either a Web Service or an application running on some device which wants to get access to the protected data or services of the resource server.

The resource owner is the person to whom the data (e.g. information, media, services) on the resource server belongs and who has the authorization to use the service and data.

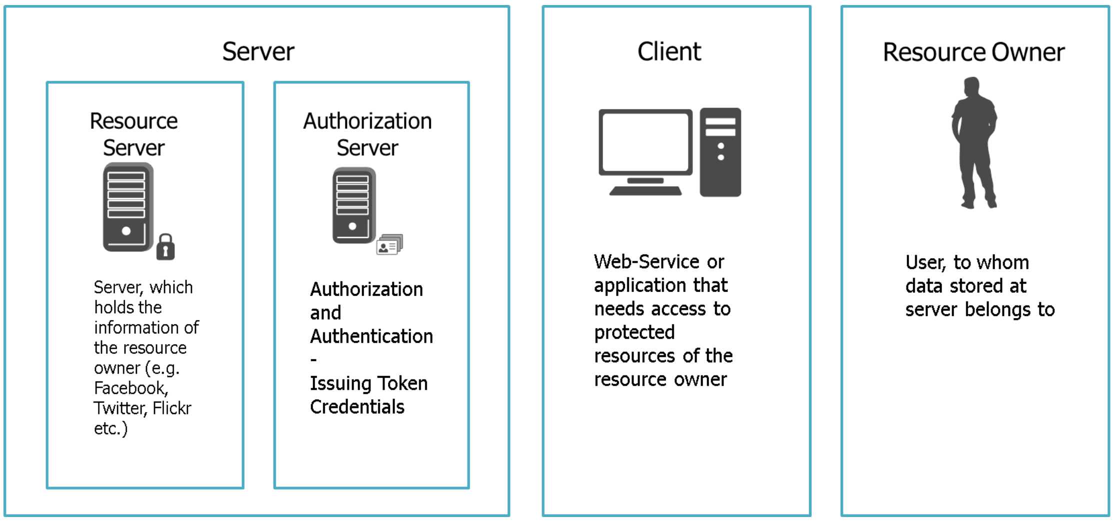

Server
Resource Server
Server, which holds the information of the resource owner (e.g. Facebook, Twitter, Flickr etc.)
Authorization Server
Authorization and Authentication - Issuing Token Credentials
Client
Web-Service or application that needs access to protected resources of the resource owner
Resource Owner
User, to whom data stored at server belongs to

Figure 2.6: OAuth Roles

Credentials

In the OAuth flow of OAuth 1.0, three types of credentials get used. The client credentials are to identify the client when sending initial requests to the server. The developer acquires these credentials by registering his application at the server. The temporary credentials are used to identify individual requests during the user authorization process and the token credentials at last are used to make authorized requests to protected resources and act on behalf of the user. In OAuth 2.0 the OAuth flow is simplified for most use cases and temporary credentials are left out, instead there is such thing as an authorization grant, which gets issued back to the client and is then used to acquire token credentials.

2.2.3 Authentication and Authorization

To understand the basic principle behind OAuth and its innovative values, it should be viewed in contrast to the traditional flow. In the traditional scenario, when a Resource Owner wants to authorize a Client to access his protected resources, he has to share his credentials with the client.

In contrary in the OAuth flow the client never actually needs to see the user credentials. To describe the OAuth 2.0 flow figure 2.9 will be used. At first the client redirects the user to the Authorization Server (A) to let the resource owner authenticate himself and authorize the client. After the authorization the user gets redirected back to the client

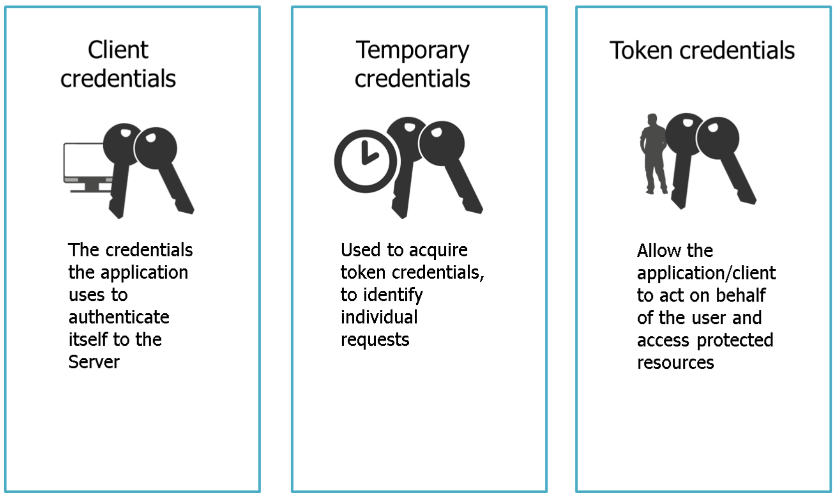

Client credentials
The credentials the application uses to authenticate itself to the Server
Temporary credentials
Used to acquire token credentials, to identify individual requests
Token credentials
Allow the application/client to act on behalf of the user and access protected resources

Figure 2.7: Credential types

with an Authorization grant (B). This Authorization Grant is then used (C) to obtain the Access Token (D) from the Authorization Server (which first checks the identity of the client and verifies the request). Now with the Access Token the Client can request resources from the Resource Server (E/F).

This is the basic OAuth flow description, since there are three types of clients that are considered, with their individual security considerations, as a result there are four different OAuth flows in OAuth 2.0.

The possible client types are web-application, user-agent based client and native application client.

The flows defined are the authorization code flows for trusted web-application clients, where the Access Token gets issued to the client backend without going through the user-agent. The implicit flow for user-agent based clients (e.g. in JavaScript in the browser), where the Access Token is directly issued after the authorization to the user-agent of the resource owner. Resource owner password credentials flow, where the password credentials can be used to acquire an Access Token, for native clients and the client credentials flow where just the client credentials are used to acquire an Access Token. The last scenario is called 2-legged OAuth, because the only parties involved are the client and the server and not the resource owner. The client by itself acquires the rights to use the API of the server. This requires trusted clients, where the Client credentials and the Access Tokens are kept confidential. In the upcoming chapters this will be an issue when it has to be applied to secure WebSocket Interfaces.

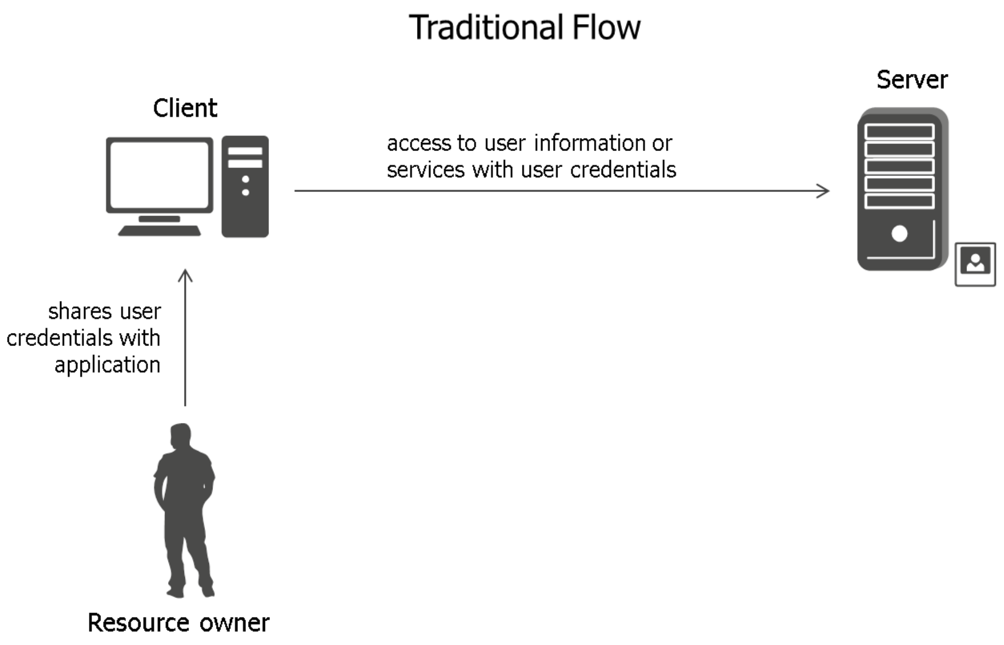

Traditional Flow
Server
Client
access to user information or services with user credentials
shares user credentials with application
Resource owner

Figure 2.8: Traditional Flow

Differences of OAuth 2.0 and OAuth 1.0

OAuth 1.0 was designed with web-application or native application use-cases in mind. The flows were rather complex and with many cryptographic requirements. In OAuth 2.0 the concept was revised and the lessons learned were applied (and still being specified) to make a more intuitive approach. Some new flows were added to give the ability for new client types to acquire authorization (like the implicit grant).

OAuth 1.0 Authentication Flow

For the explanation figure 2.10 is used. First when one goes to the site (1) of the client web application it asks you to grant access to your Twitter account. In the background it sends a request to Twitter signed with its client credentials (2) and receives back some temporary credentials (3). With the temporary credentials the client web application can redirect the user to the Authorization site of Twitter (4). The user can login with Twitter (5) and is presented with the type of access the services requests. Now the user can authorize the application or refuse (5).

When authorization is granted Twitter either can redirect back to a site (6), which was specified on registration or it can show a passphrase which will be typed in back in the calling application (6b) (for non-browser services).

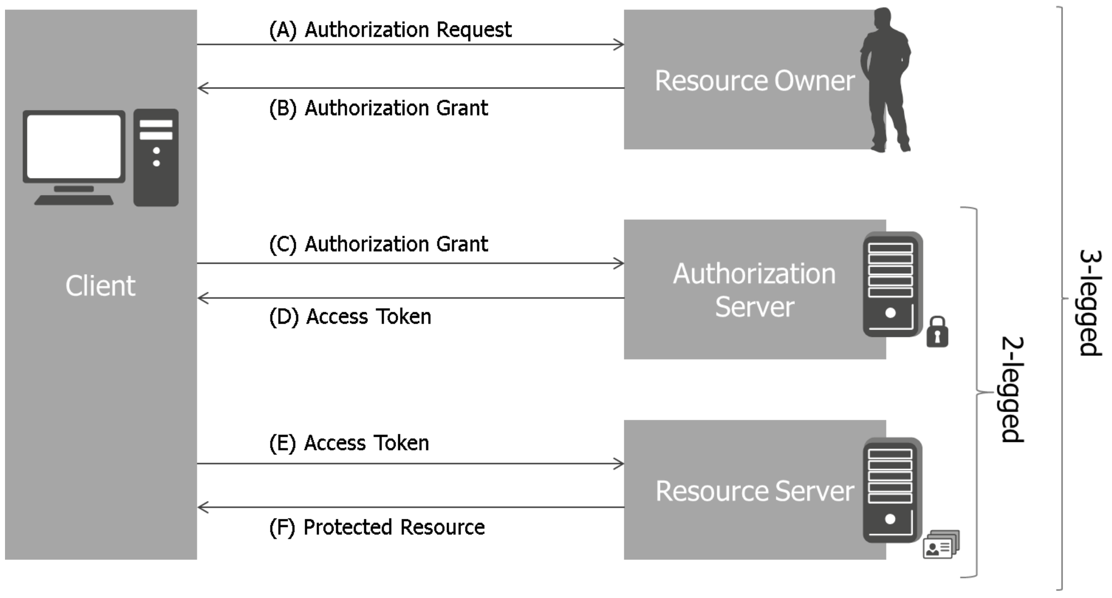


Figure 2.9: OAuth Flow

Now the client sends another request with the verified temporary credentials (7) and receives the accessToken as a response (8). With the accessToken authorized requests to the APIs can be made (9).

2.2.4 Scribe

Scribe is an Open Source OAuth Client Java library by Pablo Fernandez licensed under the MIT License [47]. It greatly simplifies the use of OAuth by handling the cryptographic signing processes and constructions of HTTP Requests. It supports many popular OAuth 1.0 and 2.0 APIs (e.g. Facebook, Twitter etc.) out of the box and makes it easy to add new ones.

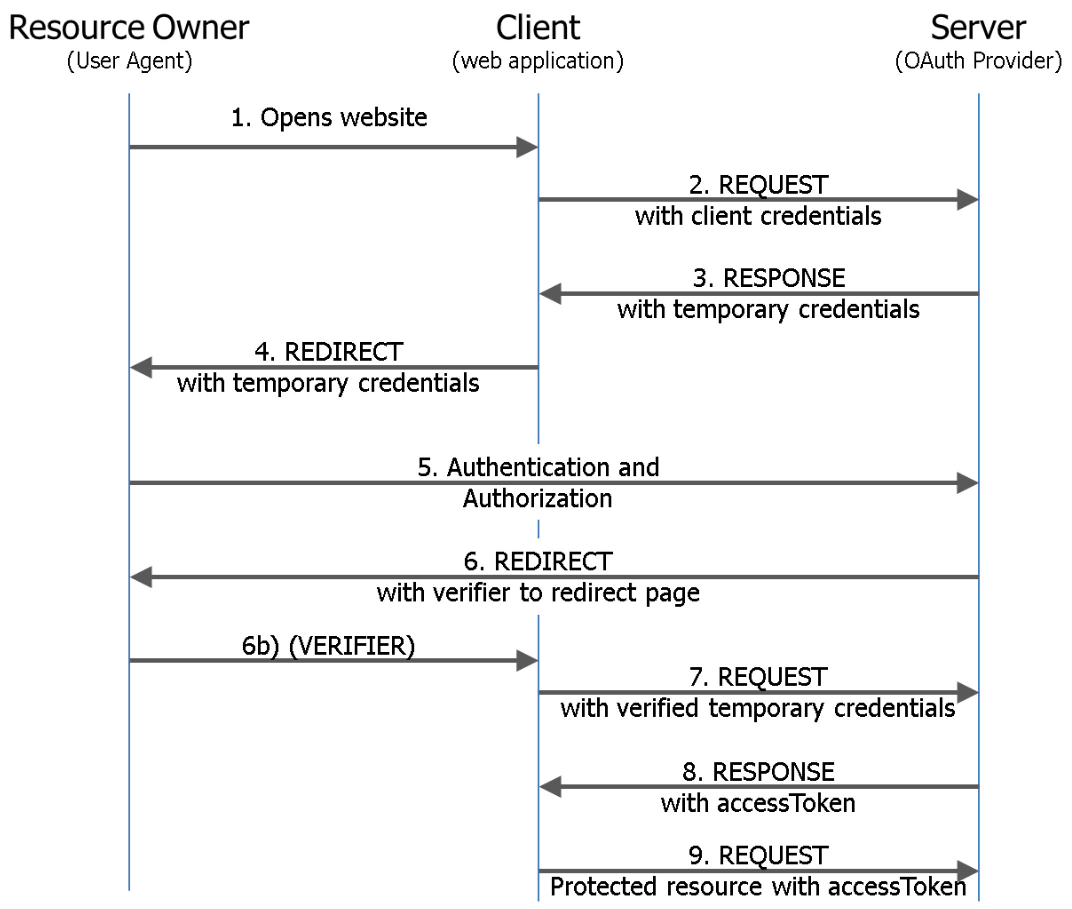


Figure 2.10: OAuth 1.0 Flow

2.3 Next Generation Network Infrastructures

This bachelor thesis is always to see in the context of Next Generation Networks (NGN), since it originated in association with the NGN Infrastructures Competence Center (NGNI) of Fraunhofer FOKUS. At this point the technologies and concepts related to that shall be explained briefly. NGNI deals with Open Service Delivery Platforms build on top of heterogeneous networks. It also provides NGN test beds for fast prototyping of new solutions for telecommunication and media companies.

2.3.1 Next Generation Networks

The goal of Next Generation Networks in the telecommunication context is achieve a convergence between all historically derived networks, where text, media or control information is sent, by treating all this information as data and sending it in packets (e.g. IP).

2.3.2 OpenIMS

The IMS is an all-IP telecom core network and an NGN. It was specified by the standardization bodies 3GPP [48] and ETSI TISPAN [49]. It is a system providing functionalities such as call control, instant messaging, presence information and location information, which all can be accessed via the Session Initiation Protocol (SIP)[9], if they are deployed in an IMS environment.

The OpenIMS [7] is an Open Source implementation of the IMS standard developed by FOKUS and used in the Open IMS Playground [50] for rapid prototyping and proof-of-concept implementation purposes.

Session Initiation Protocol

The Session Initiation Protocol [9] is the de facto standard signaling protocol for VoIP. It is standardized by the IETF. SIP will play a big role later in the development of RTC Client for the browser.

2.3.3 FOKUS Broker

The FOKUS Broker [5] is a system that enables the usage of Telecommunication Services (like SMS or Call Control) from Internet applications, by exposing these services through different kinds of Service Interfaces. It is also a service execution environment, which makes it possible to control execution through enforcement of different user and profile policies.

The services the Broker provides can be exposed through different kinds of service interfaces like REST, SOAP or RMI. These provided services are called inside-out services. Sometimes Broker services need to use third-party web service, which can be imported and are called outside-in services.

The Broker is part of the Open SOA Telco Playground of FOKUS [51], which is a testbed for telecommunication services and open service oriented architectures and is set on top of NGNs, Legacy Networks, Future Internet networks and Next Generation Mobile Networks. It is used for research in the field of "Distributed service orchestration and composition with real-time constraints, Service brokerage, Network abstraction APIs and M2M APIs, Open client infrastructures, Virtualization of network and service infrastructures in cloud-based infrastructures" [52].

Its modular architecture makes orchestration and combination of different services relatively convenient. The Broker is running in an OSGi environment, so all of the services and components are packaged as OSGi Bundles. When developing for the Broker, these bundles can be imported if their functionality is needed and exported so other services can use their capabilities. The services discussed in this thesis are also implemented as OSGi Bundles for the FOKUS Broker.

2.4 Related Work

Since WebRTC is a hot topic right now, many companies and institutions are working on use cases. Especially Ericsson[13] goes in a similar direction like the research efforts in this thesis. They already implemented a prototype SIP gateway and a Media Relay for interoperability with an IMS [53] and review possible use cases the WebRTC makes possible [54]. Their implementation architecture is depicted in figure 2.11. They also work on browser implementations and take part in the specification process at IETF.

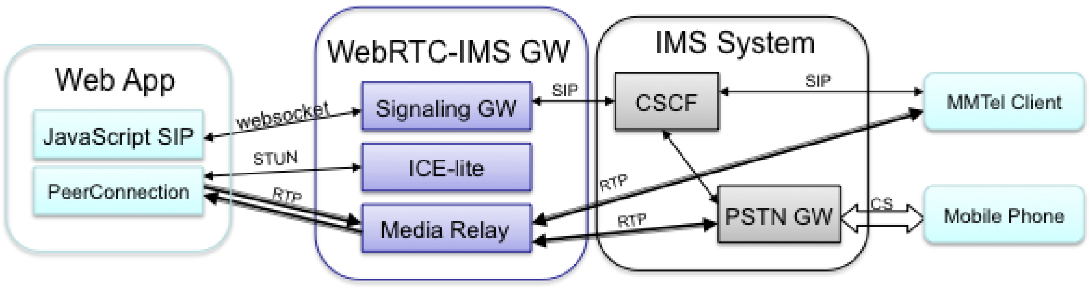


Figure 2.11: Ericssons WebRTC interoperability approach [53]

Another implementation going in a similar direction is the sipml5 [55] client by the Doubango Telecom [56]. It is an HTML5 SIP client capable of Audio/Video calls between browser clients and SIP clients (with a special profile for RTCweb). It is tested with clients on Android or iOS devices. It has also instant messaging, presence, call hold/resume and other features. The implementation consists of a SIP Proxy server called *webrtc2sip* and the actual HTML5 client. Due to its GPL license it is not suitable for commercial use.

The team of Voxeo Labs team [31] is also heavily invested in utilizing the features of WebRTC. They have several projects dedicated to bringing telco features to the web and enabling SMS, Instand Messaging and Voice calls across multiple platforms. The most significant in this segment are their PhonoSDK [57] and Tropo [58] projects. Phono is a JavaScript library, that sets the goal to turn the web browser into a phone. It can connect to SIP clients, write instant messages and making phone calls. The backend is handled by the Voxeo Cloud based backend solutions, so a web application developer does not need to take care of that. He only needs to acquire an API key from Phono so it can connect to their Web APIs. The downside is, that this API key could never be safe from exposure in a web application. Tropo is a service that can send SMS and also initiate voice calls. It can also parse speech input and can do text-to-speech when it calls someone or play a sound file. Now with WebRTC they already achieved calls from a public switched telephone network (PSTN) to the Chrome Canary web browser and are eager to build in in the new functionalities into their Phono project as fast as possible [59]. The goal of the Phono project is to make a JavaScript API that is as significant for WebRTC as jQuery for HTML and CSS.

By comparing the approach this thesis pursues with the concurrent approaches, it is noticeable that the goals are similiar. The realization, however, is different. For the SIP interworking capabilities it is not a simple proxy, what will be implemented, instead the available Broker services will be used. That gives more control, flexibility over the whole process, which we already have, through the Broker execution environment and policy engine.

2.5 Summary

In this chapter an overview of the fundamental technologies and concepts for this thesis was established. At first the topic of WebRTC was introduced with the high-level description of the APIs, the Media Path and Signaling Path were introduced with the corresponding codecs and protocols, and the current state of the development of the specifications and implementations was presented. The OAuth protocol was discussed, its advantages were shown and the different roles, credential types and flows were explained. The thesis was put in context with NGN(s) and the corresponding technologies like the IMS and the FOKUS Broker. At last the related work on this topic was presented and compared withe the aspired solution.

3 Requirements

3.1 OAuth Client Manager Service

3.1.1 Overview

The OAuth Client Manager service for the FOKUS Broker is basically a service that has to get authorization from its users for external OAuth services and provide the acquired tokens to other services inside the Broker. It is called client, because it acts as OAuth client to the external OAuth providers; and manager, because it manages and stores the received tokens from these services. The user will also have the possibility to revoke or refresh existing authorizations over a web interface. Broker services that need an authorization will have to redirect the user to the OAuth Client Manager, so they can provide the needed authorization.

3.1.2 Functional Requirements

No.	Functional Requirement	Priority	Remarks
1	Correct OAuth signing	Major	OAuth requests according to specification
2	Acquire OAuth tokens	Major	Requesting OAuth tokens, parsing the HTTP messages correctly
3	accessToken storage and mapping	Major	Mapping of users to services to tokens
4	ServiceInterface for other Broker Services	Major	Getting stored tokens of users
5	User Web Interface	Major	Interface were user can delete/refresh tokens
6	Extendable with new OAuth services	Optional	Reading configurations for other OAuth services from other configuration files
7	User Authentication	Minor	Authenticate with Broker users

Table 3.1: Functional Requirements

In table 3.1 the functional requirements for this OAuth service of the Broker are listed.

In the following they are be explained in further detail.

1/2. The OAuth signing process should be correct and according to specifications and with it the different kinds of tokens should be retrievable from all standard conform OAuth 1.0 and 2.0 services. To reduce the amount of code that has to be written for this service an extensive search on existing OAuth client implementations preferably in Java will be made.

3. When accessTokens get acquired for a particular service, they should get mapped to the service and the (Broker) user and stored (in a database or in memory) in a way that makes them easy accessible later on.

4. The ServiceInterface of this service should make it easy to get tokens from a specific user if he has already given the authorization to the Broker. If the authorization was not acquired yet it should return a link for the site where the user can give the Broker the permissions needed.

5. The User Web Interface is the site where the user can provide authorizations for all available services. He can also refresh his tokens there or delete the previously provisioned authorizations.

6. The service should be extendable with new OAuth services without changing and recompiling the application code, preferably with simple configuration files, which list the client credentials, the OAuth provider name and if necessary the OAuth endpoints (e.g. if the service is not part of the provided services).

7. Users should authenticate themselves like in the standard web interface of the Broker. From this authentication the username should be retrieved, which is then used for the mapping in 3.

3.1.3 Non-Functional Requirements

The authorization process should be convenient for the user. This means it should not have to many steps, be misleading or confusing. At the same time it should be convenient for the Broker service developer, so it is worthwhile to use the OAuth service instead of just using Scribe.

Support for some important OAuth services should be already built in (like Facebook or Twitter) and it should not be too complex to incorporate new ones.

3.2 RTC Service

3.2.1 Overview

The RTC Service uses WebRTC to enable real time communication (e.g. audio/video call and messaging) between browsers. The signaling and messaging runs over a server. Whereas the audio/video stream runs over a peer-to-peer connection. The RTC Service will also enable basic interworking with the IMS System by enabling messaging.

Due to the prototypical nature of the RTC service the requirements are more vaguely defined. The goal of this service at first is to explore the possibilities of WebRTC in combination with telecommunication services and other third-party Internet services.

3.2.2 Functional Requirements

No.	Functional Requirement	Priority	Remarks
1	Audio,Video and Chat between browsers	Major	Via peer-to-peer WebRTC connections
2	Signaling Path over WebSocket	Major	Establish a signaling path between users
3	Combination with telco services	Major	Use of telco services in the Broker
4	Combination with 3rd-party Internet services	Major	Via the OAuth service 3.1
5	SIP Interworking	Major	Sending SIP Messages from Browser Client
6	Protocol	Major	Handle communication over the Signaling Path

Table 3.2: Functional Requirements

The functional requirements listed in table 3.2 in some detail:

1. The features WebRTC introduces to web browsers should be utilized, so video, audio and chat sessions between browser clients MUST be implemented for the service.

2. The Signaling Path (see 2.1.2) for the RTC service should be done via WebSockets. Since the protocol functionalities of WebRTC are not sufficient for call control, messaging and other functionalities a protocol has to be specified to deal with these features.

3. The RTC service needs to embed some telco functionality provided by the Broker to present the possibilities that come with this new standard.

4. This service also shall be combined with the OAuth capabilities previously implemented by the OAuth Client Manager Service (3.1) and use third-party Internet services in an authorized manner to enhance the user experience.

5. Another important aspect is to enable interworking with systems like an IMS, so a first approach is to enable messaging between a SIP client and a browser RTC client.

6. A protocol needs to be defined that is then used over the WebSocket to communicate with the server and the other clients. It must support all the required functionalities.

3.2.3 Non-Functional Requirements

The non-functional requirements are, that the application running in the browser should check the browser for its capabilities, in this case for WebSocket and WebRTC. If the browser does not have those capabilities, then it should notify the user that the browser does not support the needed features or use fallback mechanisms.

Design an appealing user interface that is easy to use and gives feedback on all the actions performed by the user. It also has to be stable so it does not crash if one does not use it right.

The other requirement is that the WebSocket connection should be unaccessible to unauthorized clients or clients running outside the Broker.

3.3 Summary

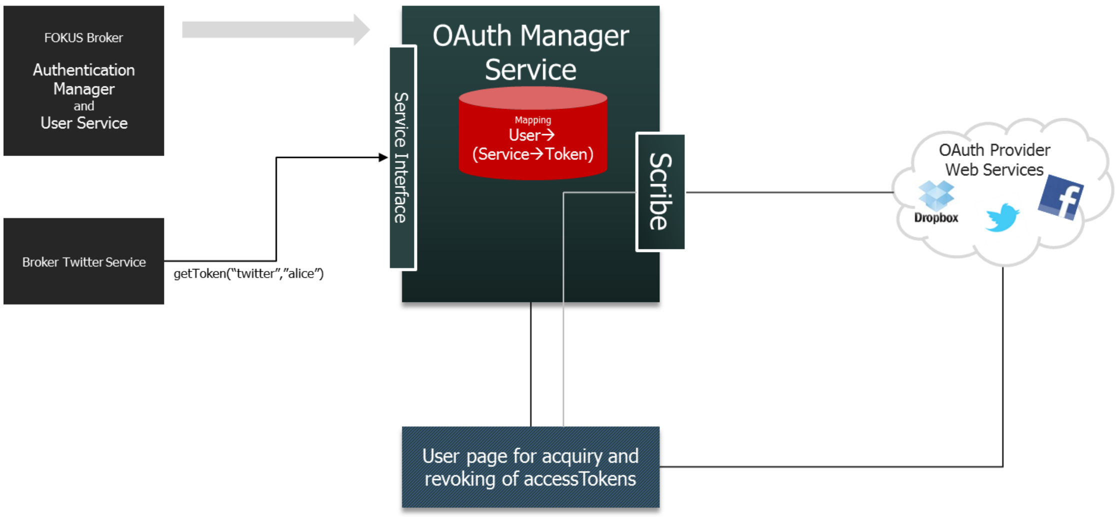
This chapter established the requirements for the OAuth service and the RTC service and client. The OAuth service requires to implement the whole OAuth process of acquiring tokens correctly and a service interface for the other Broker services to use it conveniently. The RTC service has some loosely defined requirements like telco and SIP interworking and utilization of the newly acquired OAuth capabilities. With the requirements established the next step is to propose a design for the services in the next chapter.

4 Design and Concept

In this chapter the basic ideas of the architecture of the OAuth Client Manager Service 4.1 and the RTC Service 4.2 are established and described.

4.1 OAuth Client Manager Service

A high-level overview of the architecture of this service can be seen in figure 4.1. In this figure all the main components of the OAuth Client Manager Service (OACMS) are depicted. The **Scribe** 2.2.4 library handles the OAuth signing and parsing of all the outgoing and incoming OAuth/HTTP requests. All acquired tokens for users are saved in a mapping (this could be a database or in memory). The mapping is from user to service to token, because each user can have one token for each available service. Then there is the service interface, which exposes this service functionality to other services in the FOKUS Broker (the ServiceInteface is explained further in section 4.1.1).



FOKUS Broker
Authentication Manager and User Service
Broker Twitter Service
getToken("twitter","alice")
OAuth Manager Service
Service Interface
Mapping
User→
(Service→Token)
Scribe
OAuth Provider Web Services
Dropbox
User page for acquiry and revoking of accessTokens

Figure 4.1: OAuth Client Manager Architecture

Then there is the web interface for the users, where they can acquire, update and delete their acquired tokens for individual OAuth services. This is the page users get sent to if they need to get an authorization for a service. In the final version the web interface should authenticate the user correctly using the standard Broker authentication mechanisms, for the prototype the authentication is not handled.

4.1.1 Service Interface

The service interface is the part that gets exposed to other services inside the Broker, so it has to provide only the functionality needed by other services in a simple way. The provided methods are depicted in table 4.1 and explained below.

method	arguments	return
getToken	String service, String user	(Scribe) OAuthToken
getTokenArray	String service, String user	String [tokenKey,tokenSecret]
getTokenKey	String service, String user	String tokenKey
getTokenSecret	String service, String user	String tokenSecret
getApiCredentials	String service	String [apiKey,apiSecret]
getAuthorizationUri	*empty*	String userWebInterfaceUri
getAvailableServices	*empty*	List<String>oauthServices

Table 4.1: ServiceInterface of the OAuth Manager Service

getToken(String service, String user): This method returns an Object of the class OAuthToken, which is the class for OAuthTokens in Scribe, so if a developer uses Scribe for their OAuth API requests, then it is recommended to get the token like this. If the token values are needed as String, then one of the other methods can be used (getTokenArray, getTokenKey, getTokenSecret).

getApiCredentials(String service): This service returns the API credentials for a specific service. The API credentials are needed to sign, OAuth Requests so this method is provided. But the API credentials should never be exposed to the clients/users of the Broker services, else they could be retrieved and used to masquerade other applications as Broker.

getAuthorizationUri(): This returns the URL to the user site of the OACMS, where the user can acquire/update/revoke authorization from other services. It is part of the service interface so that users can be redirected to this site, if they do not have provided the needed authorization for a service.

getAvailableServices(): This returns a List of all the available services the OAuth Manager currently supports.

4.2 RTC Service

The RTC Service is divided into the client-side code that runs in the Browser (preferably in some new Chrome dev build that supports WebRTC) and is written in JavaScript (and HTML/CSS), and the server-side code, which is the Broker service running on a Jetty 7 [60] with the SignalingServlet at its core.

4.2.1 Browser Client

The browser WebRTC Client starts a WebSocket connection with the SignalingServlet, when a user logs in. Over this WebSocket everything else is handled: session establishment, negotiation, messaging, call control and status information (e.g. online status). Therefore a proprietary JSON protocol 4.2.2 was set up as a prototype.

The Browser implementation checks for the Browser capabilities of the user. When the user does not support a specific feature like WebSocket or PeerConnection it tells that to the user. Some features MAY have fallback mechanisms.

4.2.2 Signaling Servlet

When a WebSocket connection is requested from the SignalingServlet, it authenticates the user. If the user authentication fails, then the WebSocket connection closes.

The SignalingServlet parses all the information that arrives from a user over a WebSocket and decides from the protocol information how to handle the request.

If a user wants to connect to another online user and the other user accepts, then the WebSockets of both users get mapped to each other, so signaling between them can occur and a PeerConnection can be established. If a signaling event arrives it is just passed through to the other side.

Signaling Protocol

The Browser RTC Client, that will be implemented uses the ROAP (see 2.1.2) approach for signaling. In this application the ROAP protocol is wrapped in another JSON protocol, specifically designed for this application, so messaging or call proposals or status updates can be managed. An example of such a JSON protocol message can be seen in listing 4.1.

Listing 4.1: JSON Protocol

```
{
   action: "message",
   data: { message: "Hello Alice", from: "Bob" },
   status: ""
}
```

In table 4.2 all the possible protocol states can be seen. The type column describes if this message is outgoing or incoming from the view of the browser. “Out” means it is sent by the browser client to the SignalingServlet and “in” means it comes from the SignalingServlet.

action	data	status	type
username	<username>	-	in
connect	<username>	-	out
connect	-	accept/deny	in
proposal	<remoteUsername>	-	in
proposal	-	accept/deny	out
message	{ message: <message>, from : <username>}	-	in/out
sip	{ message: <message>, to : <username>}	-	out
sip	-	true/false	in
sms	{ message: <message>, to : <username>}	-	out
sms	-	true/false	in
hangup	-	-	in/out
signaling	<ROAP/SDP>	-	in/out
update	{username, status, sip, sms}	-	in
initial	array [{username, status, sip, mobile}]	-	in
image	<base64/imagedata >	-	out
image	-	okay/unauthorized	out

Table 4.2: Protocol Overview

Here a quick description of all the actions:

- **username:** propagates the own username back to the browser, if logged in correctly
- **connect:** outgoing call to someone with the <username>
  the message returns then with a status if call got accepted or denied
- **proposal:** incoming call from someone with the <remoteUsername>
  call can get accepted or denied by the user and send back to the SignalingServlet
- **message:** incoming and outgoing chat messages

- **sip:** outgoing sip messages to specified user, returns with true if send successfully, else false
- **sms:** outgoing sms messages to specified user, returns with true if send successfully, else false
- **hangup:** if a user hangs up - tells the application to tear down the PeerConnection
- **signaling:** wraps the ROAP message and sends them to the other side, or an incoming gets unwrapped and passed down to the PeerConnection
- **initial:** after establishment of the websocket connection this array of all the users gets sent with their username, statuses and if they have a sip/mobile address
- **update:** status updates of users
- **image:** upload of base64 encoded image data to the Broker

4.2.3 IMS Interworking

What needs to be achieved is the messaging between the browser RTC Client and a SIP Client, when the other user is offline in the RTC Client. For this a special Service Profile (SP) for users in the IMS has to be configured and an Initial Filter Criteria assigned to it. What the Initial Filter Criteria (iFC) does is associating Trigger Points (TP) to application servers. Trigger Points decide which SIP packets are passed to the application specified in an iFC.

The Trigger Point is configured to pass SIP MESSAGE's to the SIP application server running in the Broker. The application server then checks to whom the message is send and if the user is currently online he sends it over the WebSocket to the RTC client in the Browser, else he answers back to the IMS with an approptiate SIP status message. A visualization of this can be seen in figure 4.2.

The other way around it works like this: The RTC user writes to an IMS/RTC user (a user with an RTC-service-profile), that is currently offline in the RTC Browser Client. The SignalingServlet realizes that the other user is offline and sends a SIP MESSAGE from the SIP Messaging Service of the Broker (if the recipient has a valid SIP mapping).

4.2.4 Basic Telco Interworking

The FOKUS Broker already provides capabilities to integrate Telco services into its applications. In this context of real time communication and messaging the SMS feature would be a useful addition. The Short Messaging Service is used similar to the SIP Messaging service from the Broker and if the user has a mobile number mapped to him, it is possible to send him an SMS.

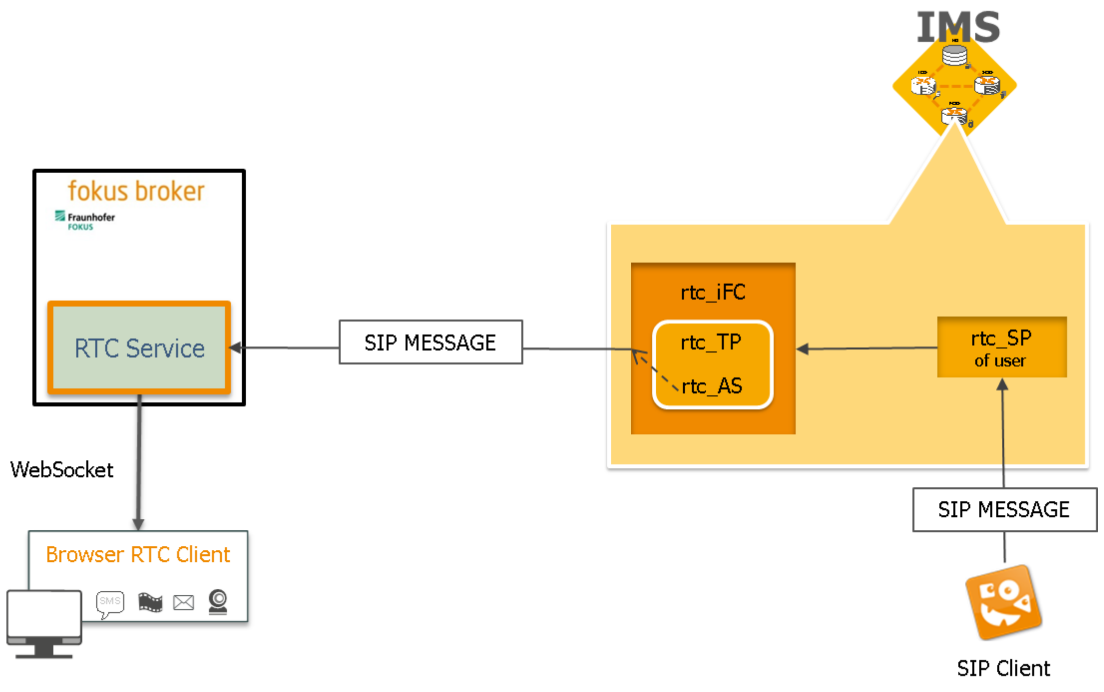


Figure 4.2: Sending Message from SIP Client to RTC Client

4.2.5 Combination with Third Party Services via OAuth

To demonstrate the combination with Third Party services via the Broker and OAuth it was decided to combine it with DropBox [61], a cloud storage and synchronizing service. The idea is to make a screenshot from the local camera video, then convert it in JavaScript to Base64 encoded image data and send it over the WebSocket to the Broker. The Broker then looks if the user has an OAuth authorization for DropBox, if this is the case it decodes the image and sends a correct OAuthRequest with the image data to DropBox.

4.3 Summary

In this chapter the overall concept is designed for both services. For the OAuth service an architecture is described. The service interface, which is designed to get used by other Broker services is defined. For the RTC Service all the components, e.g. the browser client and the signaling servlet and the intended functionalities are explained. Then the communication protocol gets defined, which is used over the WebSocket. Furthermore a concept for the incorporation between telecommunication services, IMS services and the OAuth service is proposed. On top of this designed concept the implementation phase was started. Some details of this phase are discussed in the following chapter.

5 Implementation

In this chapter some important implementation details of the OAuth service and the RTC service will be explained.

5.1 Environment

The following software, respectively operating systems, were used for the implementation:

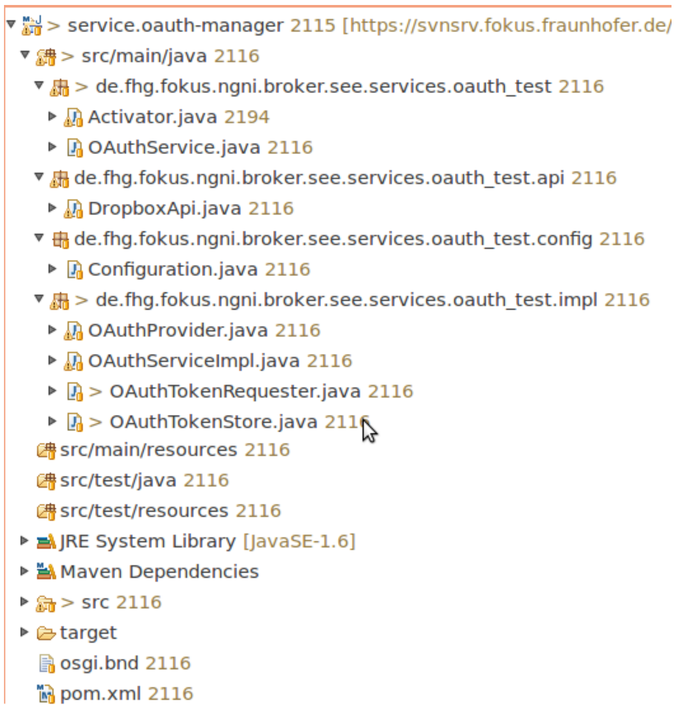
- **Execution Environment**
  - Ubuntu 11.04 64bit
  - FOKUS Broker
  - OpenIMS
  - Google Chrome dev-build(s)
  - Jetty 7.6.4
- **Development Tools**
  - Java Development Kit (JDK) 6 Update 22
  - Eclipse Indigo 3.7
  - Maven 2
- **Server Side Implementation**
  - Scribe
- **HTML, JavaScript**
  - jQuery
  - Modernizr
  - prefixfree.js

5.2 OAuth Client Manager Service

The service is split into two different projects, one of them holds all the relevant classes and exports all the features to other Broker services and is packaged as *.jar and the other one is the web interface for the users to acquire, revoke and update tokens and is packaged as a *.war. Both projects are Maven projects and OSGi compliant bundles and the OAuth web interface bundle depends on the OAuth Manager bundle. In the following section the structure of both projects is shown and the classes with their purpose are explained.

5.2.1 Project Structure

In figure 5.1 you see the structure of the OAuth Manager project.



```
> service.oauth-manager 2115 [https://svnsrv.fokus.fraunhofer.de/
  > src/main/java 2116
    > de.fhg.fokus.ngni.broker.see.services.oauth_test 2116
      Activator.java 2194
      OAuthService.java 2116
    de.fhg.fokus.ngni.broker.see.services.oauth_test.api 2116
      DropboxApi.java 2116
    de.fhg.fokus.ngni.broker.see.services.oauth_test.config 2116
      Configuration.java 2116
    > de.fhg.fokus.ngni.broker.see.services.oauth_test.impl 2116
      OAuthProvider.java 2116
      OAuthServiceImpl.java 2116
      > OAuthTokenRequester.java 2116
      > OAuthTokenStore.java 2116
  src/main/resources 2116
  src/test/java 2116
  src/test/resources 2116
  JRE System Library [JavaSE-1.6]
  Maven Dependencies
  > src 2116
  target
  osgi.bnd 2116
  pom.xml 2116
```

Figure 5.1: OAuth Manager Project Structure

Activator
The Activator class starts when the bundle is started. It loads configurations for all available OAuth services.

OAuthService
This class is the ServiceInterface specified in section 4.1.1.

OAuthServiceImpl
The actual implementation of the OAuthService interface.

DropboxApi
The Dropbox Api is not available in Scribe (unlike Facebook or Twitter and many

others), consisting of all the relevant OAuth endpoints e.g. Authorization URL, RequestToken Endpoint and AccessToken Endpoint

Configuration
The class that holds configuration properties like the URL of the OAuth service. (The only property right now)

OAuthProvider
Each instance of OAuthProvider represents an OAuth service (e.g. Twitter, FaceBook, DropBox). It consists of the API keys and secrets for the corresponding services and handles the requesting and verifying of OAuth Tokens by delegating it to the Scribe library. It has different mechanisms for OAuth 1.0 and 2.0 services. All the registered services are also saved in a Map associated to their name as String.

OAuthTokenRequester
A helper class for the web interface of the OAuth Manager for requesting and verifying tokens for users.

OAuthTokenStore
This class holds the mapping of users, services and tokens and provides the methods to retrieve or change the information.

The service had to be split into two different bundle, because the Broker/OSGi evironment does not allow to export packages from a *.war packaged file. The Web Interface with the Java Servlet Pages (JSP) is packaged in a *.war file and imports the other *.jar packaged bundle that provides all the needed packages classes and servlets.

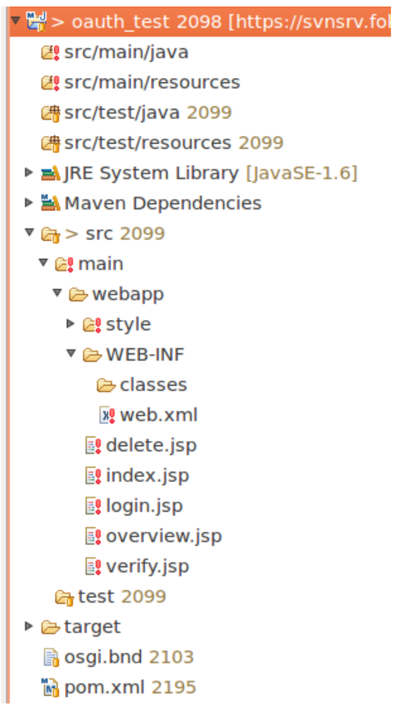
5.2.2 Web Interface

The structure of the web interface project is shown in figure 5.2.

Here a short explanation to the individual JSP's:

login.jsp
This JSP handles login of a user, although authentication is not yet implemented. One can simply log in by typing in any username. This will be the username all the tokens will be associated to and which then can be retrieved via `getToken()` from the OAuth Manager service interface.

index.jsp
Here a user can see all the services that are available in the OAuth manager service and get an authorization for them (see figure 5.3. If authorization was already provided it shows the token key and secret for that service and the user can delete it or update. If the user clicks on AUTHORIZE or `update` he is redirected to the request URL of the corre-



```
▼ > oauth_test 2098 [https://svnsrv.fol
    src/main/java
    src/main/resources
    src/test/java 2099
    src/test/resources 2099
  ▶ JRE System Library [JavaSE-1.6]
  ▶ Maven Dependencies
  ▼ > src 2099
    ▼ main
      ▼ webapp
        ▶ style
        ▼ WEB-INF
            classes
            web.xml
          delete.jsp
          index.jsp
          login.jsp
          overview.jsp
          verify.jsp
      test 2099
  ▶ target
    osgi.bnd 2103
    pom.xml 2195
```

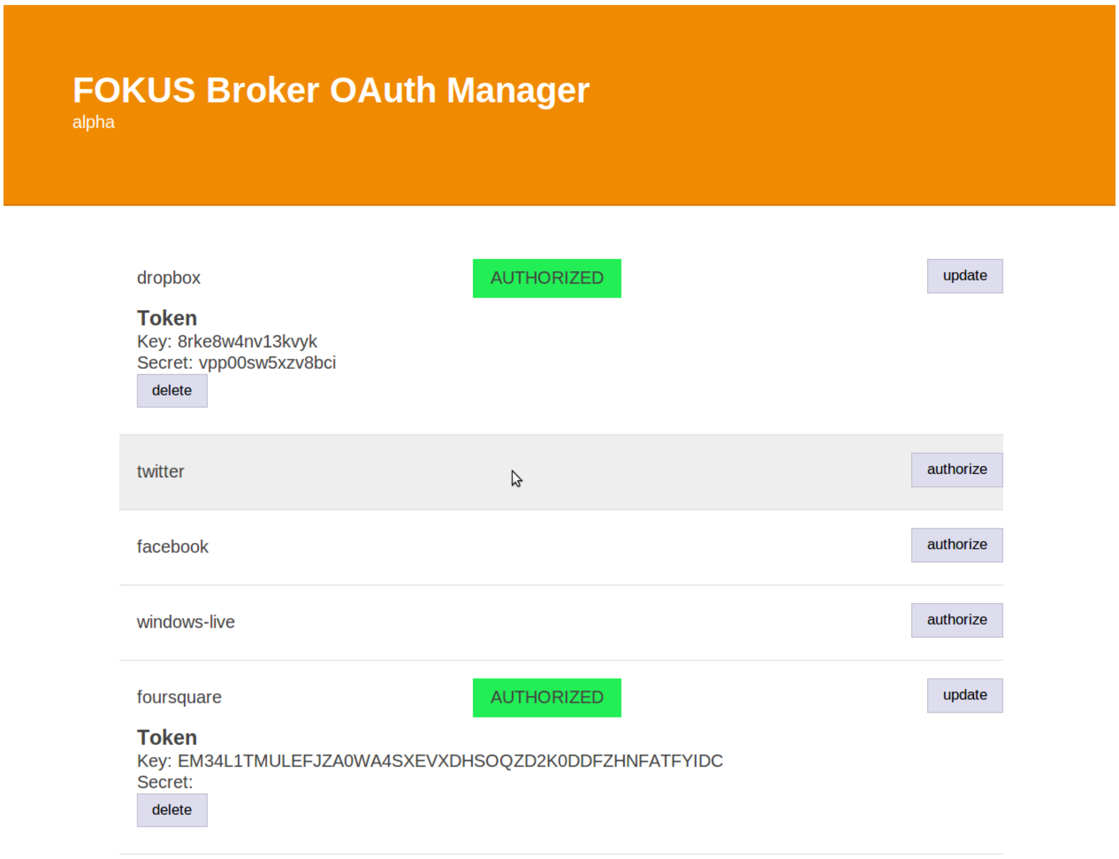
Figure 5.2: OAuth Web Interface Project Structure

sponding service. The INDEX.JSP gets the request URL from an OAuthTokenRequester object instantiated with the current username. After the user authorizes at the OAuth Provider endpoint he is either automatically redirected to the verify.jsp or he has to type in the verfier manually into the verifier field or just click verify if the user did not get a verifier.

delete.jsp
This JSP handles the deletion of tokens of a user. It is invoked with a service as parameter, when the user clicks on the delete button in the index.jsp.

verify.jsp
This JSP is either invoked with the parameters service and code through a redirect after the OAuth authorization process or if the user clicks on verify. It takes the OAuthTokenRequester object to verify the previous request and retrieve the accessToken.

overview.jsp
Shows the overview of all services, tokens and users for debugging purposes.



Token
Key: 8rke8w4nv13kvyk
Secret: vpp00sw5xzv8bci

Token
Key: EM34L1TMULEFJZA0WA4SXEVXDHSOQZD2K0DDFZHNFATFYIDC
Secret:

Figure 5.3: OAuth Web Interface

5.2.3 Dependencies

The broker.see.main and the org.osgi.core dependencies are specified in the Broker developer guide as mandatory dependencies. The Scribe library 2.2.4 is for the OAuth signing and requesting processes. Scribe was not OSGi compliant at first and had to be converted to make it possible to deploy in the Broker environment. The web interface is also dependent on the OAuth Manager service.

Listing 5.1: Dependencies of the OAuth Manager project

```
<dependencies>
  <dependency>
    <groupId>org.osgi</groupId>
    <artifactId>org.osgi.core</artifactId>
    <version>4.2.0</version>
    <type>jar</type>
    <scope>provided</scope>
  </dependency>
```

```
    <dependency>
      <groupId>de.fhg.fokus.ngni.broker.see</groupId>
      <artifactId>broker.see.main</artifactId>
      <version>1.1-SNAPSHOT</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.scribe</groupId>
      <artifactId>scribe</artifactId>
      <version>1.3.0</version>
    </dependency>
  </dependencies>
```

Listing 5.2: Additional dependency of the OAuth web interface project

```
    ...
    <dependency>
      <groupId>broker.see.services</groupId>
      <artifactId>service.oauth-manager</artifactId>
      <version>1.0-SNAPSHOT</version>
      <type>jar</type>
      <scope>compile</scope>
    </dependency>
    ...
```

5.3 RTC Service

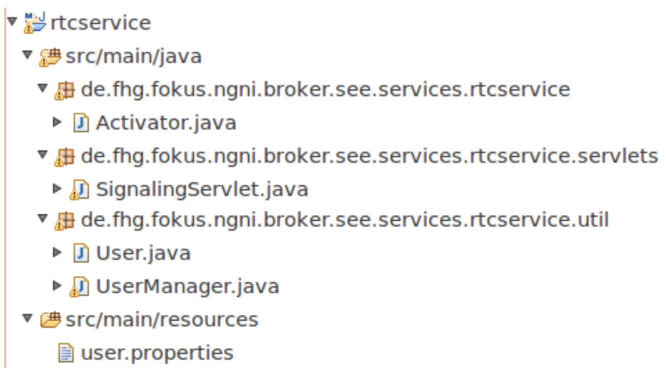
The RTC Service is seperated into the browser client and server implementation. Both are in a single Maven project that is .war-packaged and OSGi conformant, so it is easy deployable in the Broker environment. In the following section an overview of the structure will be given and the purpose of each class will be mentioned.

5.3.1 Project Structure

In figure 5.4 is the structure of the Eclipse project of the RTC service depicted.

Activator
The Activator is the class that gets executed on startup of this bundle and initializes the users in the UserManager.

SignalingServlet
The SignalingServlet is the WebSocketServlet implementation for incoming WebSocket connection requests and contains also the WebSocket implementation, which is responsible for processing incoming and sending outgoing messages. This is the most significant



```
▼ rtcservice
  ▼ src/main/java
    ▼ de.fhg.fokus.ngni.broker.see.services.rtcservice
      ▶ Activator.java
    ▼ de.fhg.fokus.ngni.broker.see.services.rtcservice.servlets
      ▶ SignalingServlet.java
    ▼ de.fhg.fokus.ngni.broker.see.services.rtcservice.util
      ▶ User.java
      ▶ UserManager.java
  ▼ src/main/resources
      user.properties
```

Figure 5.4: RTC Project Structure

class in this implementation since it handles all the protocol parsing and makes the resulting actions.

User
The User class reflects the representation of a User in the RTC service. A user in the RTC service has a username and can have a mobile number or SIP uri. He has also a status (available, busy, offline) and a WebSocket.Connection. With the WebSocket.Connection of a user a message can be sent directly to the Browser of the user. The class also sends out all the updates to all online users if a status of a user changes. The attribute connectedTo equals the username of the user this instance is currently connected to.

UserManager
The UserManager class has a Map of all the users mapped to their usernames. It has a function to retrieve these users, to add users, to check whether they exist and to remove users. There is also a function to send the information of all user statuses to a specific user (this happens when the user establishes the WebSocket connection). The UserManager also loads users from a property file, which have a mapping to a sip uri or a mobile number.

5.3.2 SignalingServlet and WebSocket

Here the operation of the SignalingServlet and SignalingWebsocket is explained in a little more detail.

Phase 1: Initialization
When the SignalingServlet gets initialized it tries to get the references of the SMS, SIP and OAuth service from the Broker.

Phase 2: WebSocket Connection Request
Incoming WebSocket connection requests invoke the doWebSocketConnect method of

the SignalingServlet. In this method the username is read from a parameter of the HttpServletRequest. Here could happen some kind of authentication, but this is not yet implemented. Instead it is just looked up if the user already exists in the UserManager and is not online and then connects with this user profile. If the user does not exist, it just creates a new user. With the User object as argument a new SignalingWebsocket instance gets created.

Phase 3: WebSocket Creation and Connection Opening
Now when the SignalingWebsocket gets created the user gets associated to that WebSocket and the WebSocket.Connection in the User object gets set. The username gets propagated back to the browser client and also all the statuses of all other users.

Phase 4: Connected
The WebSocket connection is ready and waiting for incoming messages. The possible messages are defined in the protocol 4.2. The protocols arrive as JSON String messages and get casted into a Map. Depending on the "action" field the SignalingWebsocket decides what to do with the message.

Servlet Mapping and configuration

The WebSocketServlet is mapped in the web.xml like a normal servlet in a web application(see listing 5.3).

Listing 5.3: Servlet Mapping

```
 <servlet>
   <servlet-name>SignalingServlet</servlet-name>
   <servlet-class>de.fhg.fokus.ngni.broker.see.services.rtcservice
      .servlets.SignalingServlet</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>SignalingServlet</servlet-name>
   <url-pattern>/SignalingServlet</url-pattern>
</servlet-mapping>
```

Dependencies

The project depends on (apart from the dependencies every Broker bundle has) the jetty-websocket bundle for WebSocket support and the jetty-util bundle for JSON parsing. The commons-codec bundle is used to convert the base64 encoded image into byte data. Then the service.oauth-manager is imported to acquire OAuth tokens for users and Scribe is used to send requests to the OAuth API of DropBox. The dependencies are shown in listing 5.4.

Listing 5.4: RTC Service - Dependencies

```
     <dependency>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-util</artifactId>
        <version>${jetty-version}</version>
     </dependency>
     <dependency>
        <groupId>org.eclipse.jetty</groupId>
        <artifactId>jetty-websocket</artifactId>
        <version>${jetty-version}</version>
     </dependency>
     <dependency>
        <groupId>commons-codec</groupId>
        <artifactId>commons-codec</artifactId>
        <version>1.4</version>
     </dependency>
     <dependency>
        <groupId>broker.see.services</groupId>
        <artifactId>service.oauth-manager</artifactId>
        <version>1.0-SNAPSHOT</version>
        <type>jar</type>
        <scope>compile</scope>
     </dependency>
     <dependency>
        <groupId>org.scribe</groupId>
        <artifactId>scribe</artifactId>
        <version>1.3.0</version>
     </dependency>
```

5.3.3 Browser Client

The browser client is not less complex than the SignalingWebsocket in its implementation, although the complexity is not as big as it would be if a long-polling method would be used. The central point of the implementation is the onmessage function of the WebSocket. All the arriving data from the WebSocket goes into this function, so what needs to be done is the same as in the SignalingWebsocket onMessage function. The incoming protocol messages need to be parsed and the according actions have to be executed. Responses are then sent back over the Websocket method send().

If a session between two users over the interface gets established, then a PeerConnection on both sides is created and signaling messages are sent through the SignalingPath, which is also the same WebSocket, it is just sent in the protocol with "signaling" value of the "action" attribute. On the other side the signaling is passed to the other PeerConnection and so it goes back and forth until both PeerConnections agreed on codecs and other details. Since ROAP (see 2.1.2) is used for session establishment, most of the work is done by the browser itself. The only thing the PeerConnection needs, when it is created is a STUN server for the NAT Traversal. Here the same server gets used as Google used in their demo [62]: stun.l.google.com:19302.

Upload to DropBox

For the screenshot functionality a canvas needed to be used to capture the pixels from the video element. The canvas has a function to convert the canvas into a base64 encoded image (png/jpg/gif). After the conversion the image can be sent over the WebSocket as text message to the SignalingWebsocket. Normally the WebSocket would crash because of the large message size, but the message size maximum of the connection was set to 10 Mbyte.

Capability tests

The browser implementation checks for all the capabilities that are crucial for this implementation to work. For some tests the Modernizr library is used. Modernizr is a JavaScript library, which tests for HTML5 functionalities. In this implemtation it checks for WebSockets. If WebSockets do not work in the browser, then trying to do anything with this implemtation is useless. Then there is a test for the (getUserMedia) function, it will tell on start up if the getUserMedia is implemented in browser. If this function is not available, then the local video or audio can not be accessed. The last test is for PeerConnection this is only performed if a session with someone is started. If the PeerConnection is not available, it means that the user can not see the video of the partner he is connected to.

The upside is that most of the functionalities, like writing SMS, SIP or Chat can be used if only the WebSocket feature is available, which should be implemented in most modern browsers. The screenshot upload to DropBox will work, when the WebSocket and the getUserMedia imlementations are available.

5.4 Summary

This chapter dealt with important implementation aspects of each of the services. At first all the used software libraries and tools were listed. Then the OAuth service structure and the individual components of the server-side service implementation and the web interface implementation were explained. For the RTC service an overview of the structure was given and the functionality of the SignalingServlet and WebSocket was elaborated in some detail. The function of each individual class was appointed in order to give a deeper insight into the server side implementation. The client in the browser was also described with a focus on the upload to DropBox and the capability tests.

6 Evaluation

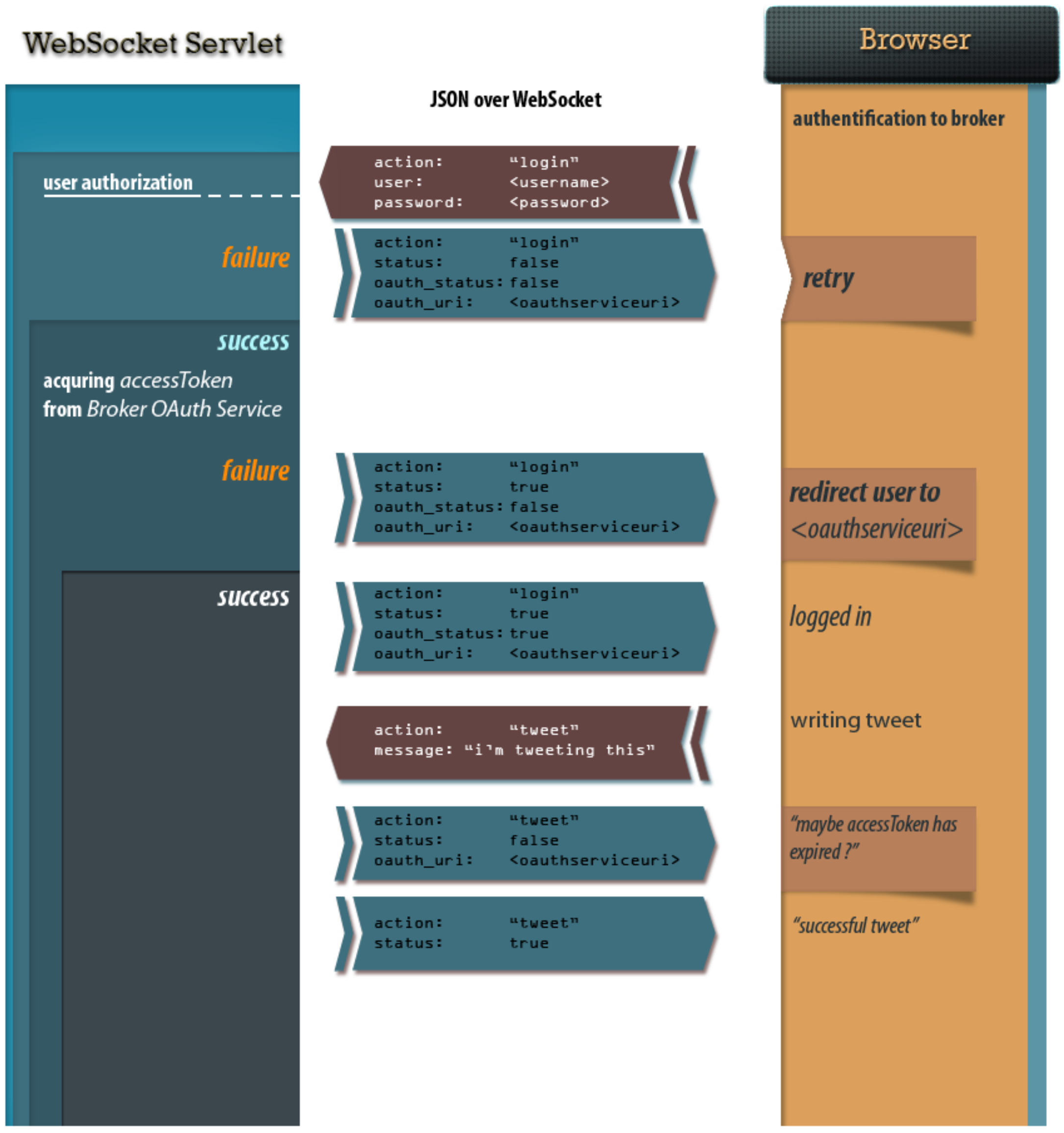
In this chapter the implementation of the OAuth Client Manager Service and the RTC Application is evaluated. Therefore it will be examined if the implementation result meets the defined requirements. After that some use cases and example services are presented were several of the features of the implemented services are combined.

6.1 Test Environment

The testing machine has the Ubuntu 11.04 64bit Operating System. Installed on the testing machine is the OpenIMS and the FOKUS Broker (see 2.3.3). The services are deployed in the Broker.

6.2 OAuth Use Cases

The first use case implemented with the new OAuth service is a simple one. What it does is to post a Tweet on Twitter (figure 6.1). The connection to the Broker service that provides this capability is done over a WebSocket. The authentication of the user is also done over the WebSocket, if it fails the connection is closed. After the authentication the Broker Service tries to get the OAuth accesstoken of the user. If this fails he sends over the websocket a redirection link to the OAuth service site, where the user can acquire a token for Twitter. When he completed the process he returns back to the Broker Twitter service site and can now send a tweet, which gets send over a websocket to the Broker and then from there with the accessToken to Twitter.



```
action:       "login"
user:         <username>
password:     <password>
```

```
action:       "login"
status:       false
oauth_status: false
oauth_uri:    <oauthserviceuri>
```

```
action:       "login"
status:       true
oauth_status: false
oauth_uri:    <oauthserviceuri>
```

```
action:       "login"
status:       true
oauth_status: true
oauth_uri:    <oauthserviceuri>
```

```
action:       "tweet"
message: "i'm tweeting this"
```

```
action:       "tweet"
status:       false
oauth_uri:    <oauthserviceuri>
```

```
action:       "tweet"
status:       true
```

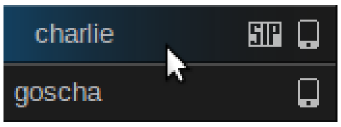
Figure 6.1: Twitter over WebSocket with OAuth

6.3 Combined RTC Use Case

In figure 6.5 to 6.11 you can see the steps of a normal WebRTC session between two browser clients. With the RTC Browser Client it is possible to start a video/audio call, between two users. At first you have to give the Browser the permission to access your Webcam. If you log in (figure 6.6), you can see all the users that are online right now and also the users that have a mobile or SIP address mapped to them. When you click on another available user it sends a session proposal to him (figure 6.7). If the user accepts (figure 6.8) then the PeerConnection gets established, which is indicated by a loading animation (figure 6.9). Now it is possible to share your webcam video with your partner by clicking on the share video switch (figure 6.10 and 6.11). A chat is also available.

In this RTC service some telco capabilities as well as the newly established OAuth capabilities are combined to enable a richer experience.

If a user has a mobile number or a SIP URI, it is shown in the sidebar. By clicking on one of these icons you can choose to write a SIP Message or an SMS to a user, even if he appears offline. If the message has been sent successfully is indicated by a notification at the bottom of the page.



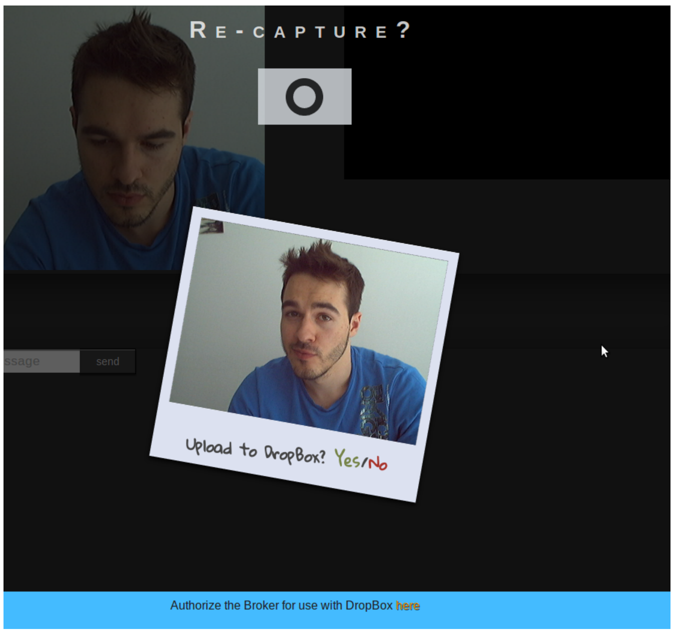
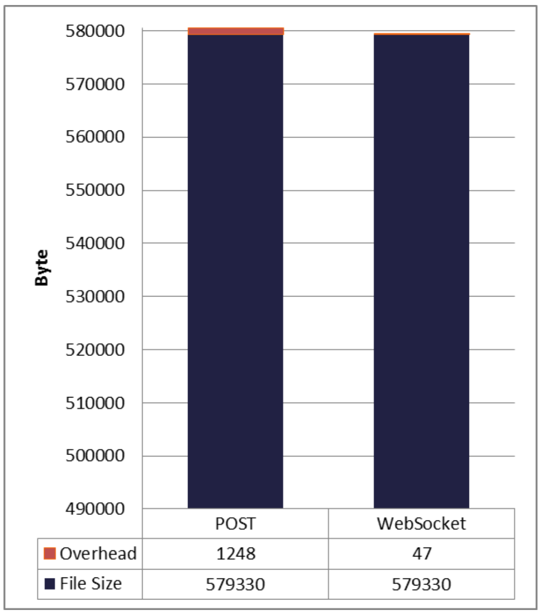

RE-CAPTURE?
ssage
send
Upload to DropBox? Yes/No
Authorize the Broker for use with DropBox here

Figure 6.2: Taking a screenshot and uploading to DropBox

Another concept here is to make it possible to make a screenshot of one of the videos and then the user has the possibility to upload the picture to DropBox (see figure 6.2). When he decides to do this the Broker RTC application requests the accessToken of this user for the service DropBox from the OAuth Manager service. If it does not exist the user gets redirected to the OAuth page of the Broker, where one has to acquire the authorization for DropBox. Then he can click again on upload and this time the image will be converted into a base64 encoded JPEG and send over the websocket to the Broker. The Broker then uploads the image with the accessToken to DropBox and in the WebInterface the notification appears that it was installed successfully.

6.3.1 Upload over WebSocket



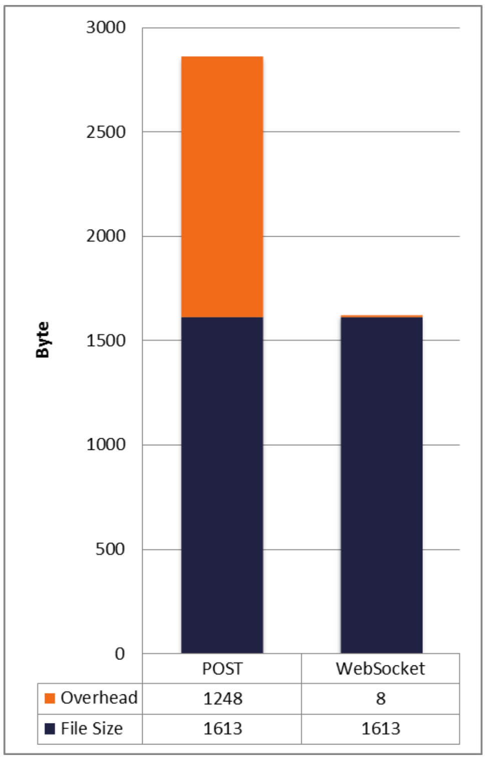
	POST	WebSocket
🟥 Overhead	1248	47
■ File Size	579330	579330

Figure 6.3: Upload Comparison - Large File

Since the WebSocket connection is already opened, the decision was made to upload the image data over the WebSocket as base64 encoded string. Now for the evaluation it would be important to see, if this upload method is faster (less overhead) than a traditional HTTP POST multipart/form-data upload.

To test, which method is more efficient a WireShark trace was made. Uploaded was a file with the size of 579330 bytes (a base64 encoded image file). The trace has shown that the multipart/form-data upload produced 580578 bytes to send over TCP. That is 1248 bytes of overhead. With the WebSocket approach on the other hand 579377 bytes were transmitted. That is only 47 bytes of overhead. For the upload duration it is not very significant, it is almost the same. It may be relevant for the server, if many users will upload something at the same time, then the large difference in the overhead will add up. In relation to the file size the overhead is not that big (see figure 6.3).

Afterwards the same tests were performed with a smaller file. A file with the size of 1613 bytes was uploaded with both methods. With the multipart/form-data the data send on the wire was 2861 bytes large, that is again 1248 bytes of overhead and this time this is a significant size increase compared to the original file size (see figure 6.4). With the WebSocket upload only 1621 bytes get sent with only 8 bytes of overhead. This is also reflected in the upload duration. With the form upload the duration is between 90 and 110 ms (with approximately 1136kBit/s upload) and the WebSocket upload is between 67 and 80 ms.



	POST	WebSocket
🟧 Overhead	1248	8
■ File Size	1613	1613

Figure 6.4: Upload Comparison - Small File

In the RTC service implementation every communication aspect (SMS, SIP, chat, upload, signaling, status), except from the audio or video stream, is sent through the WebSocket. All these messages are relatively small in size, just a few hundred or thousand bytes. In this case it is obvious that the WebSockets save an enormous amount of overhead compared to an XMLHttpRequest for example. With a rising amount of users and message throughput the overhead would increase tremendously, but with WebSockets it is just a fraction of the size it would have with Comet approaches (also see 2.1.6).

6.4 Conclusion

No.	Functional Requirement	Priority	Fulfilled
1	Correct OAuth signing	Major	yes
2	Acquire OAuth tokens	Major	yes
3	accessToken storage and mapping	Major	yes
4	ServiceInterface for other Broker Services	Major	yes
5	User Web Interface	Major	yes
6	Extendable with new OAuth services	Optional	x
7	User Authentication	Minor	x

Table 6.1: Functional Requirements - OAuth Service

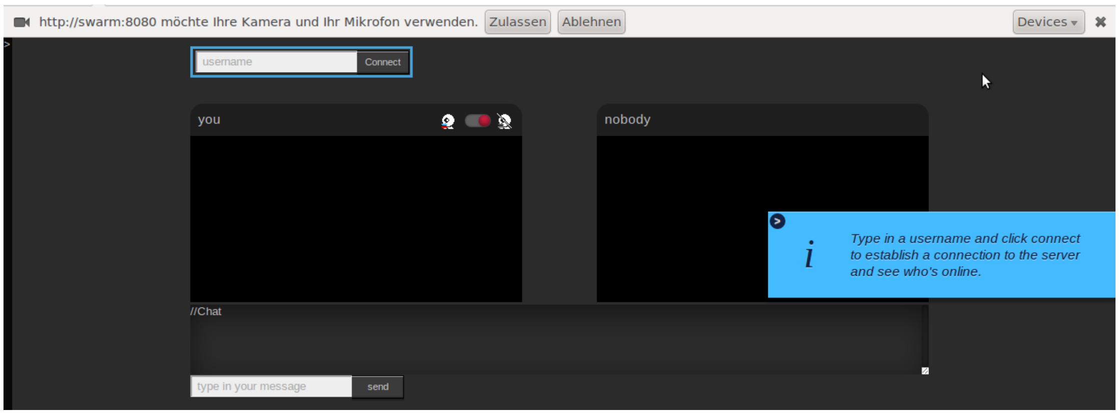
Depicted in table 6.1 and 6.2 are the defined requirements and whether there were fulfilled or not. All the major requirements defined for this thesis were accomplished. Some important requirements, like the User Authentication were defined as minor for the scope of this thesis, because the goal was to create a prototype and not a production ready implementation. All of the requirements will probably be implemented at some point afterwards.

No.	Functional Requirement	Priority	Fulfilled
1	Audio,Video and Chat between browsers	Major	yes
2	Signaling Path over WebSocket	Major	yes
3	Combination with telco services	Major	yes
4	Combination with 3rd-party Internet services	Major	yes
5	SIP Interworking	Major	yes
6	Protocol	Major	yes

Table 6.2: Functional Requirements

6.5 Summary

In this chapter the previously implemented services were evaluated. Therefore the different use cases were explained. Then the performance of a WebSocket upload compared to a normal HTTP multipart/form-data upload was examined. In conclusion it was pointed out if the achieved implementations met the proposed requirements.



Type in a username and click connect to establish a connection to the server and see who's online.

Figure 6.5: RTC Client - Permission to use camera

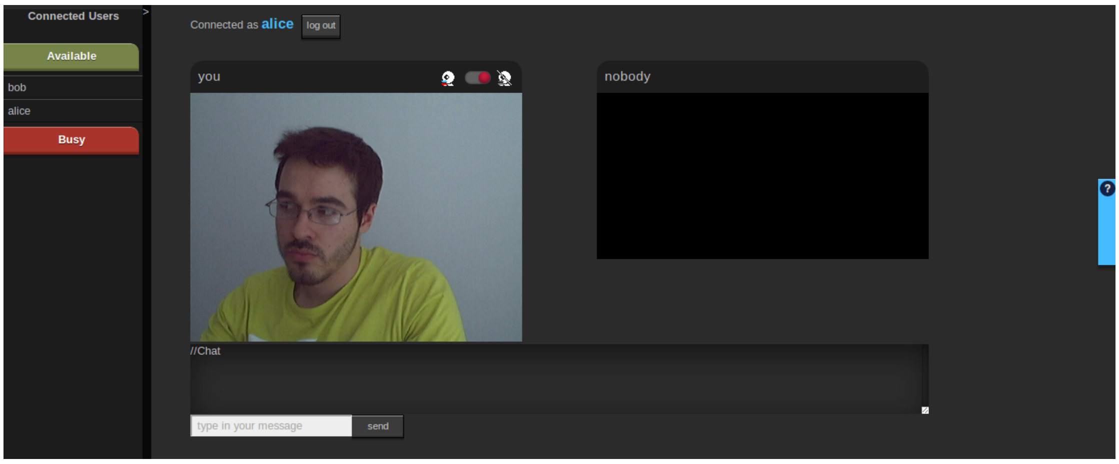
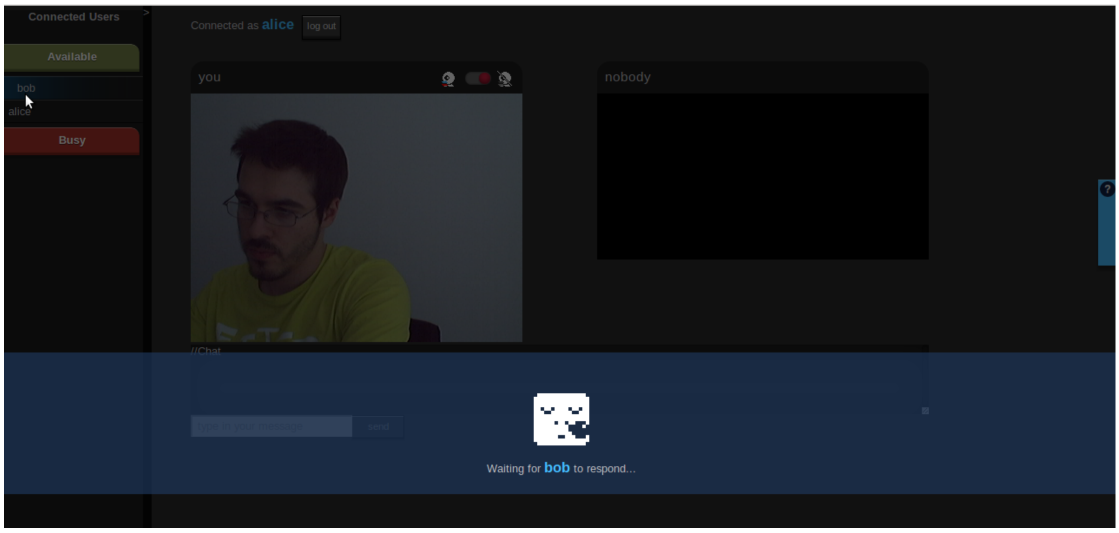
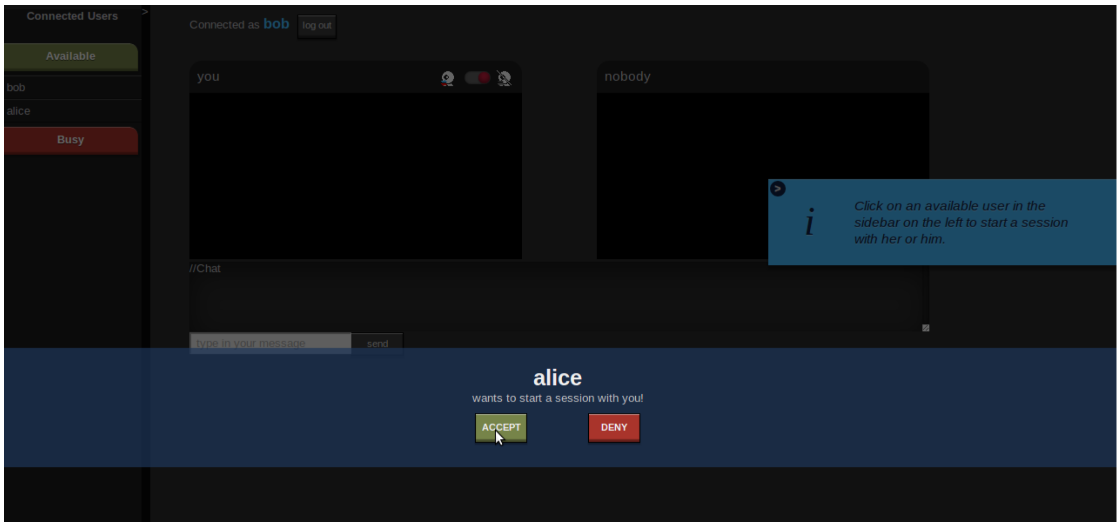


Figure 6.6: RTC Client - Logged in



Waiting for bob to respond...

Figure 6.7: RTC Client - Calling User



Click on an available user in the sidebar on the left to start a session with her or him.

alice
wants to start a session with you!

Figure 6.8: RTC Client - Getting Call

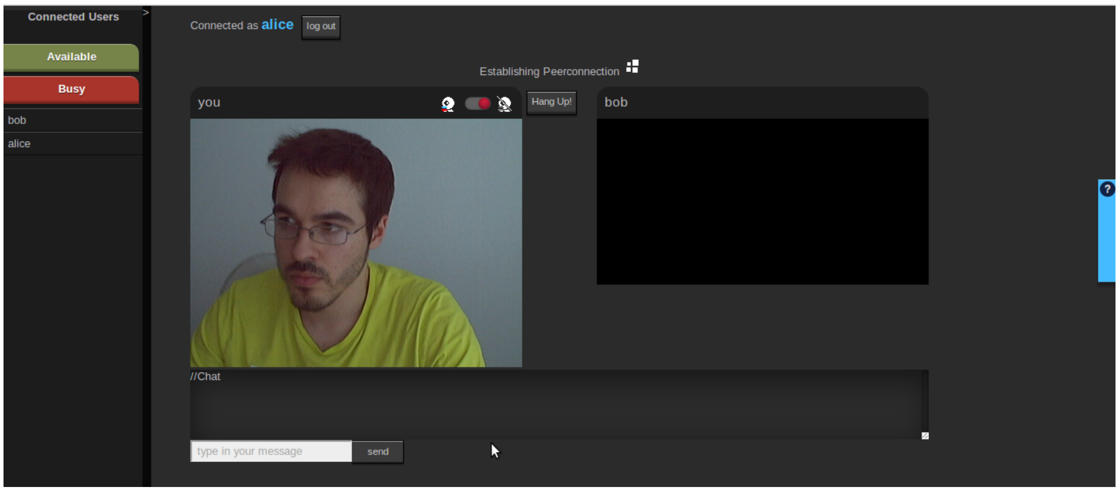
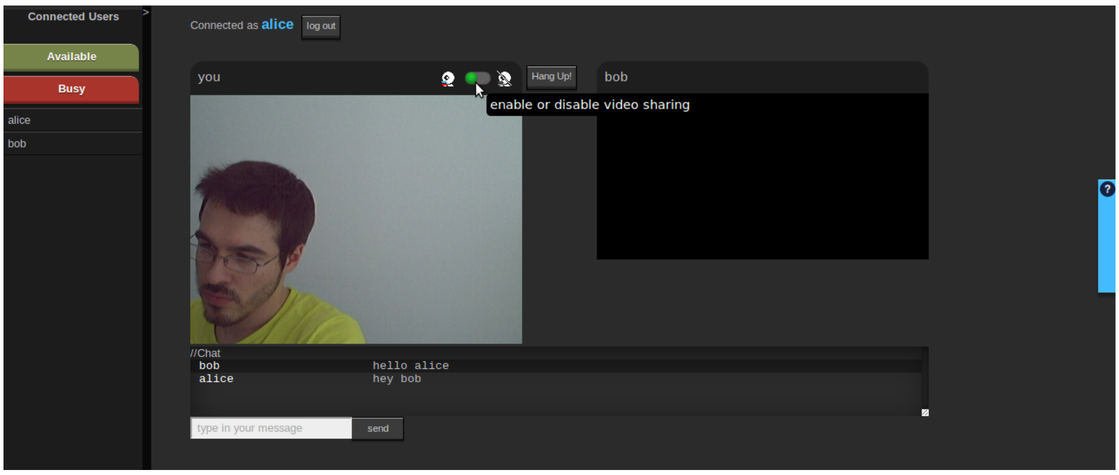
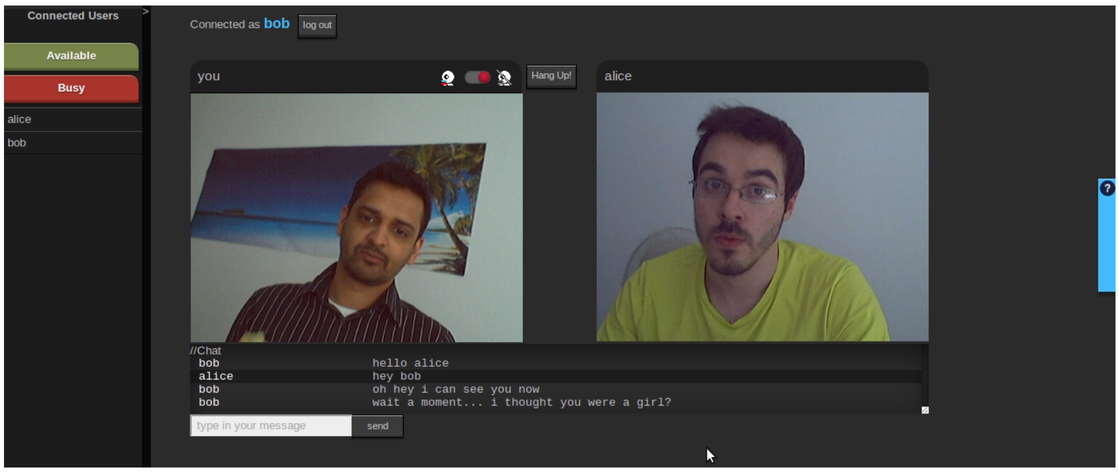


Figure 6.9: RTC Client - PeerConnection gets established



```
bob                     hello alice
alice                   hey bob
```

Figure 6.10: RTC Client - Sharing Video



```
bob      hello alice
alice    hey bob
bob      oh hey i can see you now
bob      wait a moment... i thought you were a girl?
```

Figure 6.11: RTC Client - Video shared

7 Conclusion

7.1 Summary

During the last months an extensive research on the topic of OAuth, WebRTC and WebSockets was carried out. After all relevant sources on OAuth were studied and the OAuth principles were understood, the requirements for an OAuth service in the Broker were established. During this phase also a search for already available OAuth Client Java implementations that could reduce the own implementation effort started. Scribe was found and tested out if it could be suitable for the use-cases in mind. With Scribe the design and implementation part of the OAuth service could begin.

At the same time the WebRTC standard development was watched closely and the specifications were studied. As soon as Google released their first browser builds with WebRTC functionalities, the experimentation with the functionality started. One demo [62] released by Google was studied exensively and the first prototype of an RTC demo was designed after it. Before an implementation started, the basic requirements were already loosely defined. Moving forward into the implementations and getting deeper understanding of the technologies, protocols and patterns used, the requirements and goals could be established clearer.

After the basic implementations of both services were finished the realization of use-cases began. During this phase some drawbacks of the implementations were discovered so some changes were made to APIs and protocols. One change in the protocol often influenced the client and server implementation and required some restructuring on both sides. This iterative development process continued for a few cycles until a satisfactory result could be achieved with the implementation and all the use-cases that were planned could be implemented. The incorporation of SIP and SMS took a while, although the functionalities were already present in the Broker.

Subsequent to the implementation the documentation and further evaluation was conducted. The WebSocket upload functionality was examined in more detail to evaluate the usefulness of this approach. The results verified that it is practical to use the WebSocket for frequent exchange of small messages between client and server, but the advantage decreases with message size.

7.2 Dissemination

The OAuth Client Manager component implemented is designed to be used by Broker service developers. It is supposed to reduce complexity of the OAuth process further and present a standadized way for the Broker services to save and retrieve OAuth tokens. Used in conjunction with the Scribe OAuth library it facilitates the use of OAuth services inside the Broker.

The RTC service is an early prototype of how the interoperation of the new WebRTC standard with the FOKUS Broker and its services can be achieved and how telecommunication services can be integrated with it. With its WebSocket interface it also demonstrates the usefullness of WebSockets for fast data exchange (e.g. with short latency and overhead) between server and browser as well as reduced complexity of the server-side and client-side implementation.

7.3 Problems Encountered

WebRTC

In the beginning it was not an easy task to get a WebRTC demo up and running with a PeerConnection that streams a video to another machine. It took a few days to figure out, that if the browsers do not have the exact same version it probably does not work. Then there is difference between the Chrome Canary browsers and the Chrome development builds. The Canary updates automatically, whereas the development builds stay the version you installed. Sometimes it did not work to establish a PeerConnection from an Ubuntu to a Windows machine.
Then there is a stability issue with some WebRTC functions in the browsers, due to their experimental nature they sometimes result in browser crashes.

OAuth

Scribe does not support refreshTokens in OAuth 2.0, so the OAuth Client Manager service implementation, also does not support them right now. If an accessToken is timed out and is not valid anymore, the OAuth Manager does not know of it and returns the timed out token, so the service that uses the Token has to be aware of it and redirect the user to the OAuth Manager site to renew the authorization. In the future it may be useful to build in the feature to check if the accessTokens are still valid and also support refreshTokens.
Scribe also was not an OSGi bundle so it had to be converted to be deployed in the Broker OSGi environment.

Miscellaneous

Along the way there were several problems with the development tools like Eclipse and its plugins. Maven often causes trouble with the depndency resolutions and SVN sometimes has some errors that prevent one from commiting the code one has written.
Along the way the Jetty version of the Broker had to be updated to a newer version, to support the new standard WebSocket protocol.

7.4 Future Work

The work on WebRTC in Fraunhofer has only begun and the specification and API drafts of IETF and the W3C are still young, so it is likely that a lot of changes will still be introduced. In the course of this thesis some changes already happened that may break the code in the near future. It is important to keep the implementations up to date and to keep track of further development. Also the following components should be implemented next for a better integration with OAuth and to enable more RTC functionalities with legacy and SIP clients.

7.4.1 Broker OAuth Provider

The OAuth provider in the Broker to secure its external APIs was already planned for this thesis, but due to rising complexity of the requirements after research it was set back to be done subsequently, after this thesis. A proposed architecture can be seen in figure 7.4.1.

7.4.2 Complete SIP Gateway

A SIP Gateway that translates all incoming signaling messages into SIP and sends them through to an IMS system is also the next step.

7.4.3 Media Gateway and Legacy System Interoperability

For the interoperability with legacy systems or sip clients on the media path a media gateway may be needed, therefore this will be also a further goal to achieve. This media gateway will have to act as a normal WebRTC endpoint to the WebRTC client (see 2.1.3). Some existing SIP Clients (like the myMonster client [63] of Fraunhofer FOKUS) could also implement the WebRTC PeerConnection 2.1.4.

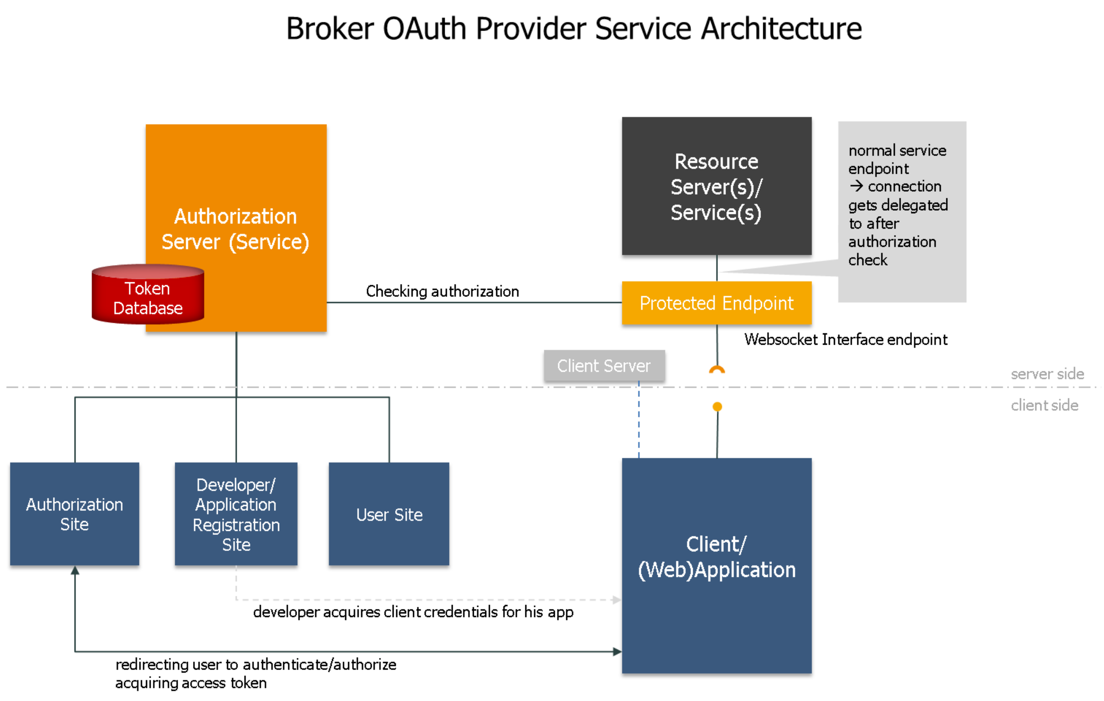

Broker OAuth Provider Service Architecture
Authorization Server (Service)
Token Database
Resource Server(s)/ Service(s)
normal service endpoint → connection gets delegated to after authorization check
Checking authorization
Protected Endpoint
Websocket Interface endpoint
Client Server
server side
client side
Authorization Site
Developer/ Application Registration Site
User Site
Client/ (Web)Application
developer acquires client credentials for his app
redirecting user to authenticate/authorize acquiring access token

Figure 7.1: OAuth Provider Architecture

List of Acronyms

3GPP	3rd Generation Partnership Project
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
AS	Application Server
CSS	Cascading Stylesheets
DHTML	Dynamic HTML
DOM	Document Object Model
FOKUS	Fraunhofer Institut fuer offene Kommunikationssysteme
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
I-CSCF	Interrogating-Call Session Control Function
IETF	Internet Engineering Task Force
IFC	Initial Filter Criteria
IM	Instant Messaging
IMS	IP Multimedia Subsystem
IP	Internet Protocol
ICE	Interactive Connectivity Establishment
JDK	Java Developer Kit
JRE	Java Runtime Environment
JSEP	JavaSript Session Establishment Protocol
JSON	JavaScript Object Notation
M2M	Machine to Machine
NGN	Next Generation Network
NAT	Network Address Translation
OMA	Open Mobile Alliance
PSTN	Public Switched Telephone Network
QoS	Quality of Service
REST	Representational State Transfer
RMI	Remote Method Invocation
ROAP	RTCweb Offer/Answer Protocol
RTC	Real Time Communication
RTP	Real Time Protocol
SDK	Software Developer Kit
SDP	Session Description Protocol
SIP	Session Initiation Protocol

SMS	Short Message Service
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
Telco API	Telecommunication API
TLS	Transport Layer Security
TP	Trigger Point
TURN	Traversal Using Relays around NAT
URI	Uniform Resource Identifier
VoIP	Voice over Internet Protocol
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Bibliography

[1] IETF: *Rtcweb Status Pages*. `http://tools.ietf.org/wg/rtcweb/`. Version: 2012. – [Online; accessed 07.03.2012]

[2] Adam Bergkvist, (Voxeo); Cullen Jennings (Cisco); Anant Narayanan (. (Ericsson); Daniel C. Burnett B. (Ericsson); Daniel C. Burnett: *"WebRTC 1.0: Real-time Communication Between Browsers", latest draft versions*. `http://www.w3.org/TR/webrtc/`. Version: 2012. – [Online; accessed 07.03.2012]

[3] Group, Web Hypertext Application Technology W.: *WHATWG*. `http://www.whatwg.org/`. Version: 2012. – [Online; accessed 15.05.2012]

[4] Fette, I. ; Melnikov, A.: *The WebSocket Protocol*. RFC 6455 (Proposed Standard). `http://www.ietf.org/rfc/rfc6455.txt`. Version: Dezember 2011 (Request for Comments)

[5] FOKUS, Fraunhofer: *FOKUS Broker Policy-based Service Access, Orchestration and Composition*. `http://www.fokus.fraunhofer.de/en/fokus_testbeds/open_soa_telco_playground/software/fokus_broker/index.html`. Version: 2012. – [Online; accessed 16.06.2012]

[6] Erl, Thomas: *What is SOA? An Introduction to Service-Oriented Computing*. `http://www.whatissoa.com/`. Version: 2012. – [Online; accessed 07.03.2012]

[7] FOKUS, Fraunhofer: *Open IMS Core Homepage*. `http://www.openimscore.org/`. Version: 2012. – [Online; accessed 16.06.2012]

[8] Wikipedia: *"Next Generation Network"*. `http://en.wikipedia.org/wiki/Next-generation_network`. Version: 2012. – [Online; accessed 07.03.2012]

[9] Rosenberg, J. ; Schulzrinne, H. ; Camarillo, G. ; Johnston, A. ; Peterson, J. ; Sparks, R. ; Handley, M. ; Schooler, E.: *SIP: Session Initiation Protocol*. RFC 3261 (Proposed Standard). `http://www.ietf.org/rfc/rfc3261.txt`. Version: Juni 2002 (Request for Comments). – Updated by RFCs 3265, 3853, 4320, 4916, 5393

[10] Larry Halff, Eran Hammer-Lahav ; Messina, Chris: *OAuth*. `http://oauth.net/`. Version: 2012. – [Online; accessed 12.04.2012]

[11] Google: *WebRTC*. `http://www.webrtc.org/home`. Version: 2012. – [Online; accessed 10.05.2012]

[12] FOUNDATION, Mozilla: *Mozilla Labs*. `http://mozillalabs.com/`. Version: 2012. – [Online; accessed 07.03.2012]

[13] *Ericsson*. `https://labs.ericsson.com/apis/web-real-time-communication/`

[14] APPLE: *HTML5 and web standards*. `http://www.apple.com/html5/`. Version: 2012. – [Online; accessed 07.03.2012]

[15] OPERA: *Opera Browser*. `http://www.opera.com/`. Version: 2012. – [Online; accessed 07.03.2012]

[16] JOBS, Steve: *Thoughts on Flash*. April

[17] JETTY & COMETD, Author: s. o.: *CometD 2.4.0 WebSocket Benchmarks*. `http://webtide.intalio.com/2011/09/cometd-2-4-0-websocket-benchmarks/`. Version: september 2011. – [Online; accessed 18.04.2012]

[18] GRECO, Peter Lubbers & F.: *HTML5 Web Sockets: A Quantum Leap in Scalability for the Web*. `http://websocket.org/quantum.html`. Version: 2011. – [Online; accessed 07.03.2012]

[19] GOOGLE: *The Chromium Projects*. `http://www.chromium.org/`. Version: 2012. – [Online; accessed 07.03.2012]

[20] ERICSSON: *WebKit library*. `https://labs.ericsson.com/apis/web-real-time-communication/downloads`. Version: 2012. – [Online; accessed 07.03.2012]

[21] WIKIPEDIA: *Comet (programming)*. `http://en.wikipedia.org/wiki/Comet_(programming)`. Version: 2012. – [Online; accessed 07.03.2012]

[22] Fraunhofer FOKUS: *Next Generation Network Infrastructures*. `http://www.fokus.fraunhofer.de/en/ngni/index.html`. Version: 2012. – [Online; accessed 07.03.2012]

[23] JENNINGS, C. ; ROSENBERG, J. ; UBERTI, J. ; JESUP, R.: RTCWeb Offer/Answer Protocol (ROAP) / IETF Secretariat. Version: Oktober 2011. `http://tools.ietf.org/html/draft-jennings-rtcweb-signaling-01`. 2011 (draft-jennings-rtcweb-signaling-01). – Internet-Draft

[24] HANDLEY, M. ; JACOBSON, V. ; PERKINS, C.: *SDP: Session Description Protocol*. RFC 4566 (Proposed Standard). `http://www.ietf.org/rfc/rfc4566.txt`. Version: Juli 2006 (Request for Comments)

[25] UBERTI, Justin: *JSEP Overview*. `http://www.ietf.org/proceedings/interim/2012/01/31/rtcweb/slides/rtcweb-0.pdf`. Version: März 2012

[26] JENNINGS, C. ; UBERTI, J.: Javascript Session Establishment Protocol / IETF Secretariat. Version: März 2012. `http://tools.ietf.org/html/draft-ietf-rtcweb-jsep-00`. 2012 (draft-ietf-rtcweb-jsep-00). – Internet-Draft

[27] Rosenberg, J.: *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols.* RFC 5245 (Proposed Standard). `http://www.ietf.org/rfc/rfc5245.txt`. Version: April 2010 (Request for Comments). – Updated by RFC 6336

[28] Rosenberg, J. ; Mahy, R. ; Matthews, P. ; Wing, D.: *Session Traversal Utilities for NAT (STUN).* RFC 5389 (Proposed Standard). `http://www.ietf.org/rfc/rfc5389.txt`. Version: Oktober 2008 (Request for Comments)

[29] Mahy, R. ; Matthews, P. ; Rosenberg, J.: *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN).* RFC 5766 (Proposed Standard). `http://www.ietf.org/rfc/rfc5766.txt`. Version: April 2010 (Request for Comments)

[30] Perkins, C. ; Westerlund, M. ; Ott, J.: Web Real-Time Communication (WebRTC): Media Transport and Use of RTP / IETF Secretariat. Version: März 2012. `http://tools.ietf.org/html/draft-ietf-rtcweb-rtp-usage-02`. 2012 (draft-ietf-rtcweb-rtp-usage-02). – Internet-Draft

[31] *Voxeo Labs.* `http://voxeolabs.com/`

[32] `http://www.cisco.com`

[33] W3C: *PeerConnection.* Working Draft. `http://dev.w3.org/2011/webrtc/editor/webrtc.html`. Version: Oktober 2011

[34] Burnett, Daniel C. ; Narayanan, Anant ; Hickson, Ian: *getusermedia.* Working Draft. `http://dev.w3.org/2011/webrtc/editor/getusermedia.html`. Version: April 2012

[35] W3C: *PeerConnection.* Working Draft. `http://dev.w3.org/2011/webrtc/editor/webrtc-20111004.html#peerconnection`. Version: Oktober 2011

[36] Ericsson: *Web Real Time Communication.* `https://labs.ericsson.com/apis/web-real-time-communication/`. Version: 2011. – [Online; accessed 07.03.2012]

[37] Google: *WebRTC Code and API – Architecture.* `http://www.webrtc.org/reference/architecture`. Version: 2012. – [Online; accessed 10.05.2012]

[38] `http://www.webrtc.org/running-the-demos`

[39] Waldon, Rick: *JavaScript: WebRTC in Opera Mobile 12.* Version: Februar 2012. `http://weblog.bocoup.com/javascript-webrtc-opera-mobile-12/`

[40] *Kaazing.* `http://kaazing.com/`

[41] Hammer-Lahav, E.: *The OAuth 1.0 Protocol.* RFC 5849 (Informational). `http://www.ietf.org/rfc/rfc5849.txt`. Version: April 2010 (Request for Comments)

[42] Hammer-Lahav, Eran: *Beginner's Guide to OAuth.* `http://hueniverse.com/oauth/guide/intro/`. Version: 2011. – [Online; accessed 07.03.2012]

[43] Hammer-Lahav, Eran: *Beginner's Guide to OAuth.* http://hueniverse.com/oauth/. Version: 2011. – [Online; accessed 25.05.2012]

[44] Hardt(Microsoft), E. Hammer/D. Recordon (.: *The OAuth 2.0 Authorization Protocol.* http://tools.ietf.org/html/draft-ietf-oauth-v2-23. Version: january 2012. – [Online; accessed 07.03.2012]

[45] *OpenID.* http://openid.net/

[46] Google: *Google Accounts Authentication and Authorization.* https://developers.google.com/accounts/docs/OpenID. Version: 2011

[47] *MIT License.* http://www.opensource.org/licenses/mit-license.php

[48] 3GPP: *Third Generation Partnership Project.* http://www.3gpp.org/. Version: 2012. – [Online; accessed 07.03.2012]

[49] ETSI: *Telecoms and Internet converged Services and Protocols for Advanced Network (TISPAN).* http://www.etsi.org/tispan/. Version: 2012. – [Online; accessed 07.03.2012]

[50] FOKUS, Fraunhofer: *Open IMS Playground.* http://www.fokus.fraunhofer.de/en/fokus_testbeds/open_ims_playground/index.html. – [Online; accessed 16.06.2012]

[51] *FOKUS Open SOA Telco Playground.* http://www.fokus.fraunhofer.de/en/fokus_testbeds/open_soa_telco_playground/index.html

[52] FOKUS, Fraunhofer: *An Open Technology and Applications Testbed for Next Generation Service Platforms and Smart Cities.* http://www.fokus.fraunhofer.de/en/fokus_testbeds/open_soa_telco_playground/home/at_a_glance/index.html. – [Online; accessed 15.06.2012]

[53] Sandgren, Nicklas ; Mecklin, Tomas ; Mäenpää, Jouni: *WebRTC Interworking with Traditional Telephony Services.* Version: Februar 2012. https://labs.ericsson.com/developer-community/blog/webrtc-interworking-traditional-telephony-services

[54] Eriksson, Göran A. ; Hakansson, Stefan: *WebRTC: enhancing the web with real-time communication capabilities.* Version: April 2012. http://www.ericsson.com/res/thecompany/docs/publications/ericsson_review/2012/er-webrtc-html5.pdf

[55] Doubango Telecom: *sipml5.* https://code.google.com/p/sipml5/

[56] *Doubango Telecom.* http://www.doubango.org/

[57] *Phono.* http://phono.com/. Version: 2012

[58] *Tropo.* https://www.tropo.com/home.jsp. – [Online; accessed 07.03.2012]

[59] *PhonoSDK WebRTC Preview.* `http://phono.com/webrtc`

[60] *Jetty.* `http://www.eclipse.org/jetty/`

[61] *DropBox.* `www.dropbox.com`

[62] `https://apprtc.appspot.com/`

[63] *myMONSTER - Telco Communicator Suite.* `http://www.monster-the-client.org/`. – [Online; accessed 21.05.2012]